

Resumen Final de Orga 1

The FurfiOS Corporation

Diciembre 2020

Índice general

1. Unidad de control y Microprogramación	3
1.1. Introducción	3
1.2. Problemas a resolver	5
1.3. Diseño de una computadora: pasos necesarios	5
1.4. Diseñando una Unidad de Control para la arquitectura MIPS	6
1.4.1. Paso 1: Analizar el conjunto de instrucciones para determinar los requerimientos del camino de datos.	8
1.4.2. Paso 2: Selección de componentes electrónicos	10
1.4.3. Paso 3: Construcción del camino de datos según los requerimientos con las componentes seleccionadas	12
1.4.4. Paso 4: Analizar la implementación de cada instrucción para determinar las señales de control necesarias	16
1.4.5. Paso 5: Construir la unidad de control que implemente el comportamiento necesario	17
1.5. Implementaciones de la Unidad de Control:	20
1.5.1. ROM	20
1.5.2. PLA	21
1.6. Microprogramación	22
1.7. Ventajas y Desventajas	24
2. Entrada / Salida	25
2.1. Introducción	25
2.2. Métodos de acceso a E/S	27
2.2.1. Puertos dedicados	27
2.2.2. Mapear a Memoria	28
2.2.3. Puertos vs Mapeo a Memoria	28
2.3. Métodos de control de E/S	30
2.3.1. Programmed Input / Output: Polling	30
2.3.2. Interrupciones	31
2.3.3. DMA	34
3. Entrada / Salida: Conversión de señales	37
3.1. Introducción	37
3.2. Información analógica	37
3.3. Amplificador operacional	38
3.4. Conversiones Analógico-Digital y Digital-Analógico	39
3.4.1. Conversión Digital a Analógica	39
3.4.2. Conversión de Analógica a Digital	40
4. Memoria y Cache	42
4.1. Introducción	42
4.2. Tipos de Memorias	43

4.2.1. ROM	43
4.2.2. RAM	44
4.3. Estructura de bus clásica	47
4.3.1. Estructura de bus con cache	49
4.4. Organización de la Cache	50
4.5. Esquemas de Mapeo	52
4.5.1. Mapeo Directo o de Correspondencia Directa	52
4.5.2. Mapeo Completamente Asociativo	53
4.5.3. Mapeo Asociativo por Conjuntos de N vías	53
4.6. Políticas de reemplazo de contenido	54
4.7. Políticas de Escritura	54
4.8. Cache Multinivel	55
5. Buses	56
5.1. Introducción	56
5.2. Buses	57
5.3. Diseño de un bus	59
5.3.1. Ancho del bus	59
5.3.2. Tipo de línea	59
5.3.3. Temporización	60
5.3.4. Arbitraje	62

Este resumen fue hecho en base a las clases teóricas de Charly del Segundo Cuatrimestre 2020, complementado con la bibliografía de la materia:

- Tanenbaum (Modern Operating Systems) [1]
- Tanenbaum (Structured Computer Organization) [2]
- Kaufmann [3]
- Null [4]
- Stallings [5]

El link para editar el overleaf es el siguiente <https://www.overleaf.com/7718548219vxdwcqcgtpvb>.

Capítulo 1

Unidad de control y Microprogramación

1.1. Introducción

Habíamos paseado por el Nivel 0, donde discutimos cómo se implementaba electrónicamente distintos tipos de circuitos combinatorios y secuenciales, que nos permiten construir funciones booleanas y memorias.

Después nos fuimos a Nivel 2, donde discutimos qué era una ISA, es decir, qué era un lenguaje de programación para una computadora de estas características a bajo nivel.



Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

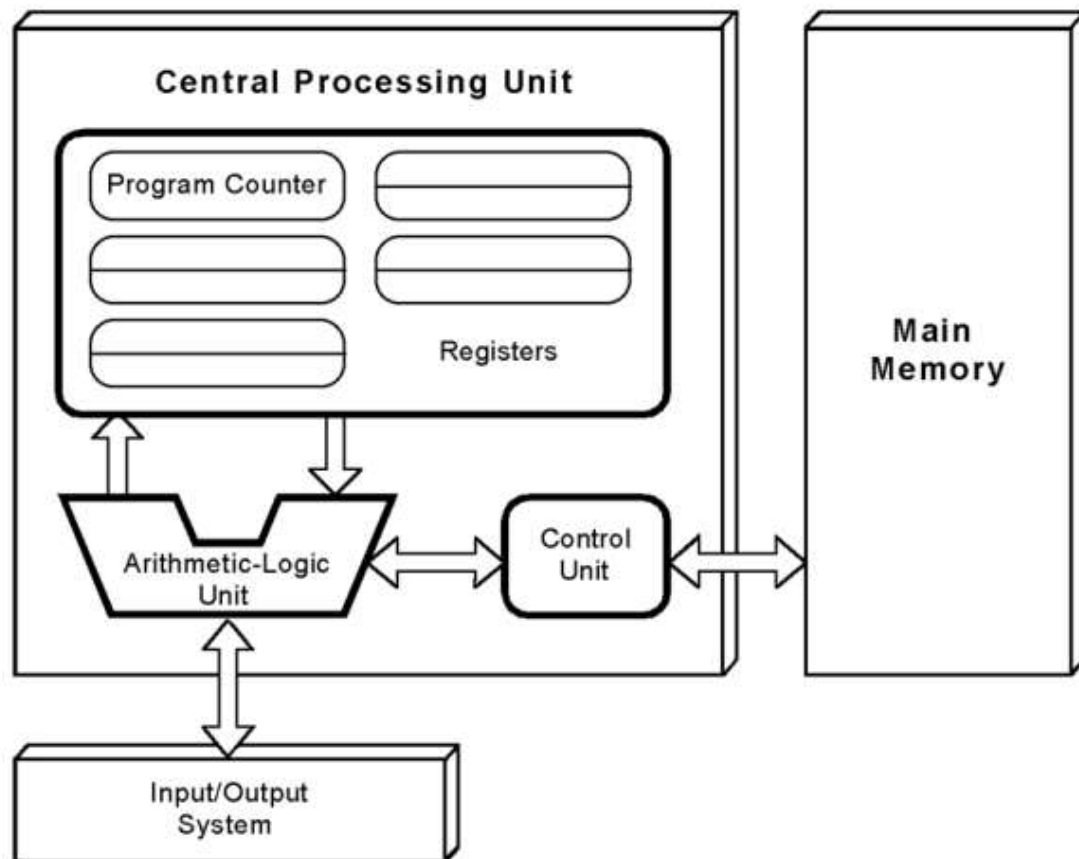
Ahora, vamos a establecer el vínculo entre ese nivel 0 y nivel 2. Es decir, vamos a ver cómo, a partir de tener una tira de bits guardada en memoria, se puede realizar un proceso de ejecución de esa tira de bits, interpretándolas como instrucciones específicas.

Recordemos que nos mantenemos en el modelo de cómputo de Von Neumann - Turing:

- Los programas y los datos se almacenan en la misma memoria sobre la que se puede leer y escribir
- La operación de la máquina depende del estado de la memoria
- El contenido de la memoria es accedido a partir de su posición
- La ejecución es secuencial (a menos que se indique lo contrario)

Entonces, tenemos una CPU que lee tiras de bits de la memoria, las cuales van a ser interpretadas como instrucciones de un programa, o como datos a ser utilizados, que ejecuta instrucciones de forma secuencial.

Recordemos que nuestra computadora tiene la siguiente forma:



Luego, tenemos 3 componentes principales:

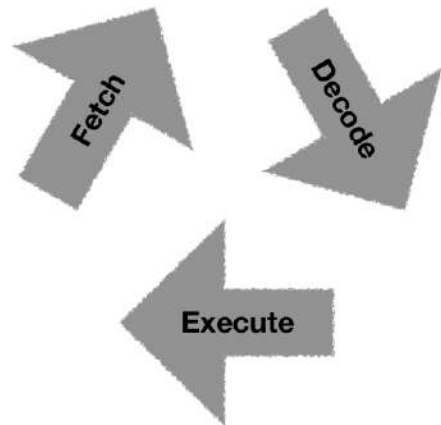
- CPU
- Memoria
- Sistema de E/S

Por lo tanto, nuestro objetivo es poder traer instrucciones de la memoria a la CPU, y ejecutarlas con datos que, o bien vienen en la instrucción, o tenemos que ir a buscarlos a memoria (ver modos de direccionamiento).

También recordemos que, cuando se tiene una computadora bajo el modelo de cómputo de Von Neumann - Turing, la ejecución es secuencial, y se asume que existe una labor cíclica que se conoce como **ciclo de instrucción**:

—>Ciclo de instrucción (UC):

Fetch	<ul style="list-style-type: none">- Se obtiene la instrucción apuntada por el PC de la Memoria- La ALU incrementa el PC
Decode	<ul style="list-style-type: none">- Se decodifica la instrucción- Se obtienen los operandos de la Memoria y se los coloca en Registros
Execute	<ul style="list-style-type: none">- La ALU realiza la operación- Se coloca el resultado en la Memoria



Vamos a comprender un programa como una tira de instrucciones, o sea que físicamente vamos a tener en la memoria las instrucciones una detrás de la otra, de manera tal que se pueda ejecutar este ciclo de instrucción.

La unidad de control se encarga de secuenciar la ejecución de programas. Pensemos que tenemos nuestra memoria por un lado, la ALU por el otro, y nuestros programas están almacenados en la memoria. La labor principal de la unidad de control es la realización de este ciclo de instrucción.

1.2. Problemas a resolver

Cuando estamos por diseñar una computadora, lo más importante es la construcción de la unidad de control. Ya habíamos visto cómo construir componentes que permiten evaluar funciones booleanas y almacenar valores, y por otro lado vimos cómo construir un set de instrucciones que nos interese poder ejecutar. Ahora tenemos que poder conectar una cosa con la otra. Entonces, ser el puente entre la ISA y la circuitería es la labor de la **unidad de control**.

Entre otras cosas, lo que va a tener que hacer la unidad de control es resolver el problema de cómo interconectar los distintos recursos compartidos (ALU, memoria, banco de registros, buses) resolviendo los problemas de contienda.

Sabemos que las instrucciones utilizan recursos que son compartidos, y por lo tanto hay que arbitrar el acceso a estos recursos compartidos, ya que una instrucción puede utilizar un mismo recurso más de una vez, haciendo que se pierdan los valores anteriores.

De esto se encarga la unidad de control al secuenciar las tareas para la ejecución de una instrucción, separando en el tiempo las tareas que compiten por un mismo recurso compartido.

1.3. Diseño de una computadora: pasos necesarios

Cuando uno tiene que diseñar una computadora, tiene que llevar a cabo varias tareas. Una manera de construir un conjunto de tareas que dan como resultado una computadora es la siguiente:

Necesitamos analizar el conjunto de instrucciones que vamos a utilizar, porque esas instrucciones nos van a determinar los requerimientos del **camino de datos**, es decir, qué datos pasan de qué recurso a qué recurso. Ese camino de datos, entre recursos que van a ser utilizados, nos dice cómo va a terminar siendo, en la práctica, la ejecución de la instrucción.

Luego, vamos a tener que seleccionar las **componentes electrónicas** que vamos a utilizar, y construir efectivamente el **camino de datos**, que las interconecta, en base a los requerimientos de las mismas.

Por otro lado, debemos analizar las **señales de control** que me permiten disponer de estas interconexiones. Por ejemplo, la ALU es un recurso que es utilizado por muchas instrucciones, pero además hay instrucciones que la utilizan con datos que provienen de distintos lugares.

Por ejemplo, se podría tener una instrucción que me sume el contenido de dos registros, y el resultado me lo coloque en otro registro. Entonces, la ALU tiene que tener dos entradas (la entrada A y la entrada B), y tengo que poder mandar lo que tengo en el banco de registros a esas entradas. La ALU, al mismo tiempo, debe poder ser utilizada para sumar el contenido de un registro con un operando en memoria. Entonces, se necesitan señales de control para poder controlar de dónde la ALU va a tomar los datos, y qué operación tiene que efectuar. Por último, construimos la unidad de control.

En resumen, tenemos el siguiente esquema:

- Analizar el conjunto de instrucciones para determinar los requerimientos del camino de datos.
- Selección de componentes electrónicos.
- Construcción del camino de datos según los requerimientos con las componentes seleccionadas.
- Analizar la implementación de cada instrucción para determinar las señales de control necesarias.
- Construir la unidad de control que implemente el comportamiento necesario.

1.4. Diseñando una Unidad de Control para la arquitectura MIPS

Vamos a trabajar sobre una arquitectura específica, con registros de propósito general, más sencilla que la Máquina ORGA 1, para simplificar el problema de construir una unidad de control. Esta arquitectura cuenta con:

- Procesador RISC (es decir, un set de instrucciones sencillo)
- 32 registros de propósito general (salvo excepciones)
- 3 tipos de instrucciones de 32 bits (tamaño fijo, lo cual facilita la construcción del ciclo de instrucción) con el siguiente formato:

Type	format (bits)						-31-	-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)		
I	opcode (6)	rs (5)	rt (5)	immediate (16)				
J	opcode (6)	address (26)						

Vemos que tenemos instrucciones cuyos operandos son:

- Tipo R (Ariteméticas): 3 registros
- Tipo I (Operaciones con inmediato): 2 registros y un inmediato de 16-bits
- Tipo J (Saltos): 1 dirección de 26-bits

Notemos que las direcciones de esta máquina son de 32-bits. Los primeros 4 (31...28) dividen la memoria en distintos segmentos (Reservado, Text, Data y Stack), que permanecen fijos a lo largo de la

ejecución de un programa. De los 28 restantes, solo los primeros 26 determinan la dirección efectiva, ya que la memoria con la que trabaja el procesador MIPS está fragmentada por bytes (tiene direccionamiento a byte), pero como quiero acceder por palabra, alineamos toda la memoria a 32 bits, haciendo que los 2 bits menos significativos sean siempre 00.

El ciclo de instrucción de este procesador MIPS, en general (algunas instrucciones no tienen algunas de estas etapas), tiene 5 etapas:

- Fetch de instrucción (**Fetch** / **IF**)
- Decodificación de instrucción (y lectura de registros si tengo que usar registros) (**Decode** / **ID**)
- Ejecución o cálculo de dirección de memoria (**Execution** / **Ex**)
- Lectura a datos de la memoria (si tengo que hacer una lectura adicional a memoria) (**Mem**)
- Escritura de resultados en la memoria o registros (**Write back** / **WB**)

Este set de instrucciones es demasiado extenso (para el marco de la materia), por lo que solo vamos a implementar 6 instrucciones del mismo (Suma, Resta, Salto, Branch, Load y Store), pero es de interés saber qué otras instrucciones ofrece esta arquitectura (para darse una idea de qué necesita una computadora real).

El set de instrucciones de esta arquitectura tiene varios tipos de instrucciones, que sirven para distintas cosas. Hay un bloque de instrucciones que sirven para carga y guardado de información. Las más básicas son:

- Load Word: Carga una palabra, es de formato I, el opcode es 35, toma dos registros y un offset.
- Store Word: Guarda una palabra, es de formato I, el opcode es 43, toma dos registros y un offset.

Ejemplo de uso:

- $LW \$1, 100(\$2) \iff Reg1 = [Reg2 + 100]$ (carga en el registro 1 el dato apuntado por el contenido del registro 2 + 100 de offset).
- $SW \$1, 100(\$2) \iff [Reg2 + 100] = Reg1$ (guarda en la dirección de memoria registro 2 + 100 el contenido del registro 1)

Por otro lado, tenemos instrucciones aritméticas, como por ejemplo:

- **ADD**: es de formato R, el opcode es 32, toma tres registros (destino, operando1, operando2).
- **Set on Less Than**: setea un valor en un registro, según la comparación entre dos registros, es de formato R, el opcode es 42, toma tres registros (destino, operando1, operando2).
- **ADDI**: es de formato I, el opcode es 8, toma dos registros (destino, operando) y un inmediato.
- **SLL**: Shift lógico a izquierda, es de formato R, el opcode es 0, toma dos registros y un inmediato.
- **MULT**: es de formato R, el opcode es 24, toma dos registros (y el resultado se encuentra en dos registros de uso específico T0 (parte menos significativa) y T1 (parte más significativa)).

Ejemplo de uso:

- $ADD \$1, \$2, \$3 \iff Reg1 = Reg2 + Reg3$
- $SLT \$1, \$2, \$3 \iff Si Reg2 < Reg3 entonces Reg1 = 1, sino Reg1 = 0$
- $ADDI \$1, \$2, 100 \iff Reg1 = Reg2 + 100$
- $SLL \$1, \$2, 4 \iff Reg1 = Reg2 * 2^4$
- $MULT \$1, \$2 \iff T0 = (Reg1 * Reg2)[31:0], T1 = (Reg1 * Reg2)[63:32]$

Y por último, tenemos instrucciones de salto:

J: es de formato J, y toma un inmediato

Ejemplo de uso: J 10000 \iff PC[27:0] = 1000000 (alineamiento)

Una vez vistas las particularidades de la arquitectura con la que trabajamos, continuaremos con el diseño de esta computadora. Para esto, seguiremos el esquema de trabajo visto en la sección anterior.

1.4.1. Paso 1: Analizar el conjunto de instrucciones para determinar los requerimientos del camino de datos.

RTL

Para poder construir las señales con las que vamos a trabajar, el camino de datos, y construir la unidad de control vamos a utilizar un lenguaje llamado RTL (Register Transfer Language) (es un pseudocódigo que usamos para abstraernos del nivel 0, y que sea más fácil pensar cómo hacemos para que las instrucciones funcionen como esperábamos. Después hay que traducirlo a señales concretas). Utilizamos este lenguaje ya que, para poder manejar el uso de recursos compartidos, colocamos pequeños registros que vamos a usar para recibir información y almacenarla temporalmente, para usarla a lo largo de la ejecución (sin perder estos valores).

Entonces, el hecho de tener varios registros a lo largo de nuestro camino de dato, nuestra interconexión de componentes, nos va a determinar la posibilidad de expresar el proceso de ejecución de una única instrucción a partir de hacer fluir información entre registros. La idea de estos registros es que funcionen como buffers, permitiendo traer información de un lugar y colocarlo en uno de estos buffers, para luego decidir a donde mandar al dato.

Entonces, el lenguaje que vamos a utilizar nos permite describir el orden en el que van a ocurrir las cosas a lo largo de la ejecución de la instrucciones, eligiendo de qué registro a qué registro va el dato, a través de las interconexiones y temporalidades necesarias para poder ejecutar una instrucción.

Veamos cómo funciona un Fetch de una instrucción (en MARIE). Por un lado tiene que traer de memoria el bloque de 32-bits indicado por la dirección que tengo en el PC, y guardarlo en el Instruction Register (IR). Por el otro lado, tengo que actualizar el valor del PC para que apunte a la próxima instrucción:

1. $IR \leftarrow [PC]$
2. $PC \leftarrow PC + 4$

Ahora sí podemos empezar a construir la unidad de control. Vamos a trabajar con un conjunto de instrucciones aún más reducido (a fines didácticos):

Ahora vamos a describir de qué manera se ejecutan estas instrucciones en lenguaje RTL:

ADD y SUB: ADDU/SUBU rd, rs, rt

Fetch:	IR	$\leftarrow [PC]$
	PC	$\leftarrow PC + 4$
Decode:	A	$\leftarrow rs$
	B	$\leftarrow rt$
Execution:	ALUOut	$\leftarrow A \pm B$
Write back	rd	$\leftarrow ALUOut$

Lo primero que tenemos que hacer es la etapa 1, el **Fetch**. Esta trae de la memoria la instrucción a ejecutar, y la coloca en el IR. Guardarlo en el IR es necesario para poder liberar la utilización de la

memoria. Luego, actualiza el PC para que apunte a la siguiente instrucción.

En la etapa de **Decode**, mirando del opcode, sabemos que estamos haciendo una suma/resta, y por lo tanto colocamos en A y B los datos con los que vamos a operar. A y B son registros específicos de la ALU, y se utilizan para poder elegir de dónde viene el dato que va a ingresar a la ALU. Esencialmente, en esta etapa, miro qué instrucción tengo, y dependiendo de cuál tengo, elijo de dónde vienen los datos, y qué operación voy a hacer.

En la etapa de **Execute**, la ALU va a ejecutar, y va a colocar en un registro específico llamado ALUOut el resultado de esta operación. La razón de la existencia de este registro es que quiero poder capturar este valor para la siguiente etapa (**Write back**), en donde se decide a dónde tiene que ir a parar.

Para poder decidir a dónde tiene que ir, primero hay que guardarlo, y luego configurar a dónde va a parar ese valor, y luego enviar ese valor. Notemos que tenemos una contienda en los recursos, ya que podríamos leer y escribir sobre el mismo registro, y resolvemos esta contienda dividiendo este proceso en dos etapas.

LOAD: LW rt, rs, imm16

Fetch:	IR	$\leftarrow [\text{PC}]$
	PC	$\leftarrow \text{PC} + 4$
Decode:	A	$\leftarrow \text{rs}$
	B	$\leftarrow \text{rt} (*)$
Execution:	ALUOut	$\leftarrow \text{A} + \text{sing_ext}(\text{IR}[15:0])$
Mem	MDR	$\leftarrow [\text{ALUOut}]$
Write back	rt	$\leftarrow \text{MDR}$

STORE: SW rt, rs, imm16

Fetch:	IR	$\leftarrow [\text{PC}]$
	PC	$\leftarrow \text{PC} + 4$
Decode:	A	$\leftarrow \text{rs}$
	B	$\leftarrow \text{rt} (*)$
Execution:	ALUOut	$\leftarrow \text{A} + \text{sing_ext}(\text{IR}[15:0])$
Mem	[ALUOut]	$\leftarrow \text{B}$

(*) Notemos que no es necesario cargar rt en B, pero lo hacemos para que la etapa de Decode sea exactamente la misma que en ADD y SUB, facilitando la lógica de la unidad de control.

El MDR (Memory Data Register) es un registro específico en el que colocamos el dato apuntado por la dirección de memoria. La idea es la misma que con el IR, solo que el MDR se utiliza cuando buscamos en memoria un dato en vez de una instrucción. Además, en el caso de tener que hacer indirecciones, esta etapa se extendería para hacer una segunda búsqueda a memoria.

Branch: BEQ rt, rs, imm16

Fetch:	IR	$\leftarrow [\text{PC}]$
	PC	$\leftarrow \text{PC} + 4$
Decode:	A	$\leftarrow \text{rs}$
	B	$\leftarrow \text{rt}$
Execution:	ALUOut	$\leftarrow \text{PC} + \text{sing_ext}(\text{IR}[15:0]) \ll 2$
	Comp(A, B)	if zero then
	PC	$\leftarrow \text{ALUOut}$

Jump: J imm26

Fetch:	IR	$\leftarrow [PC]$
	PC	$\leftarrow PC + 4$
Execution:	PC	$\leftarrow PC[31:28] ++ (IR[25:0] \ll 2)$

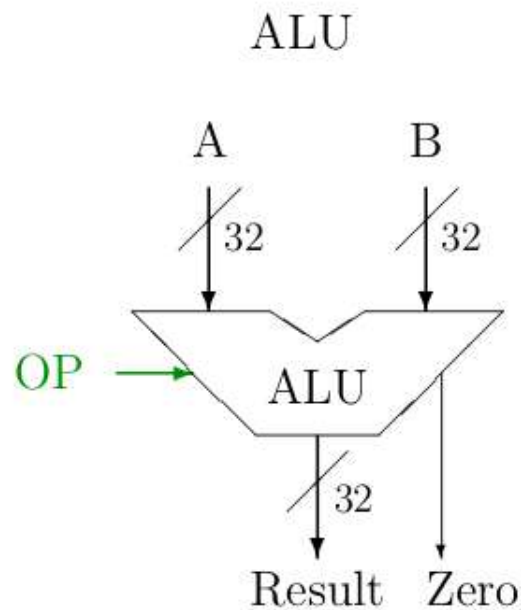
Una vez tenemos el código RTL de todas las instrucciones, nos podemos armar la siguiente tabla:

Cycle	Instruction type	Action
IF	Todas	$IR \leftarrow Mem[PC]$ $PC \leftarrow PC + 4$
ID	Todas salvo J	$A \leftarrow R[rs]$ $B \leftarrow R[rt]$ $ALUOut \leftarrow PC + sign_ext(IR[15..0]) \ll 2$
EX	ADDU/SUBU LW/SW BEQ J	$ALUOut \leftarrow A + - B$ $ALUOut \leftarrow A + sign_ext(IR[15..0])$ Comp A B, if zero then $PC \leftarrow ALUOut$ $PC[31..2] \leftarrow PC[31..28] ++ (IR[25..0] \ll 2)$
Mem	LW SW	$MDR \leftarrow Mem[ALUOut]$ $Mem[ALUOut] \leftarrow B$
WB	ADDU/SUBU LW	$R[rd] \leftarrow ALUOut$ $R[rt] \leftarrow MDR$

Acá podemos ver para qué nos sirve hacer micro-operaciones de más en algunas instrucciones. En todas las instrucciones, salvo las de tipo J, tenemos la mismas etapas de **Fetch** y **Decode**, simplificando la lógica de control.

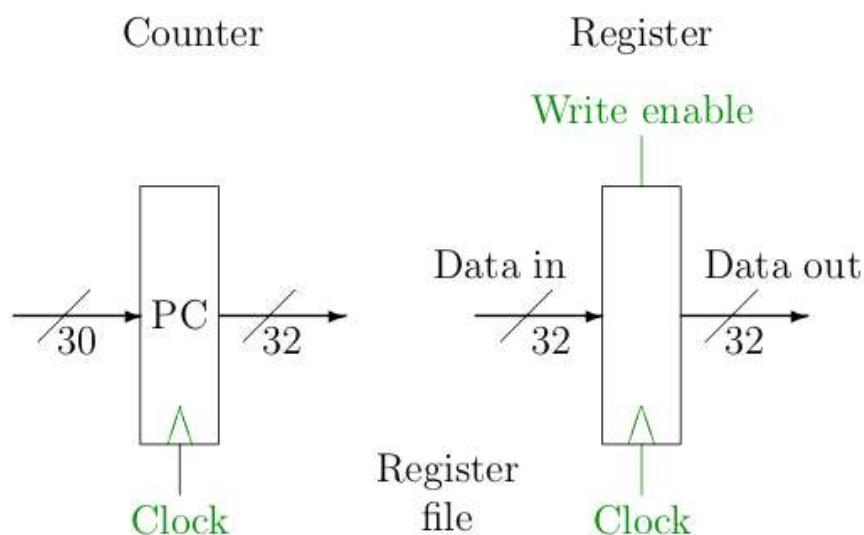
1.4.2. Paso 2: Selección de componentes electrónicos

Ahora, vamos a elegir las componentes con las que vamos a trabajar. Por un lado, vamos a tener una ALU de 32-bits, con dos entradas: A y B, y dos salidas: ALUOut (que me devuelve el resultado de la operación) y Zero (que me indica si una comparación dio zero).

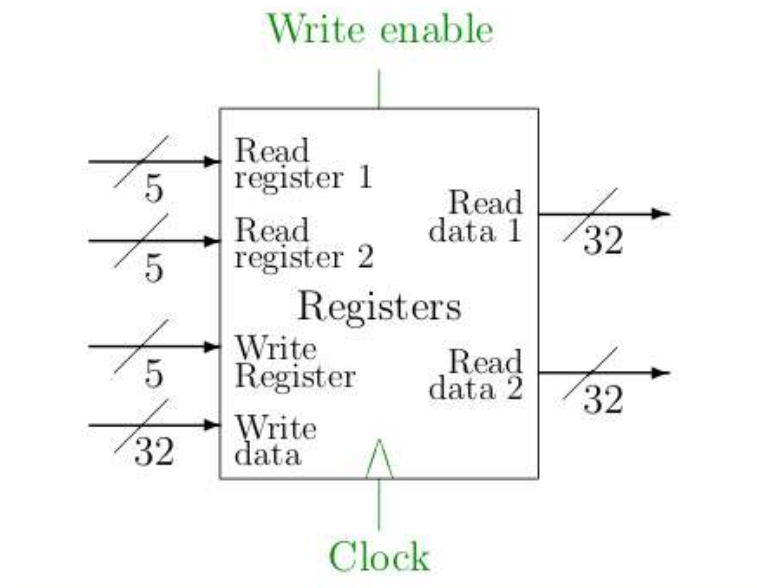


Además, como los valores de entrada para la ALU dependen del código de operación (por ejemplo $PC + \text{sign_ext}(\dots)$) se necesitan multiplexores adicionales.

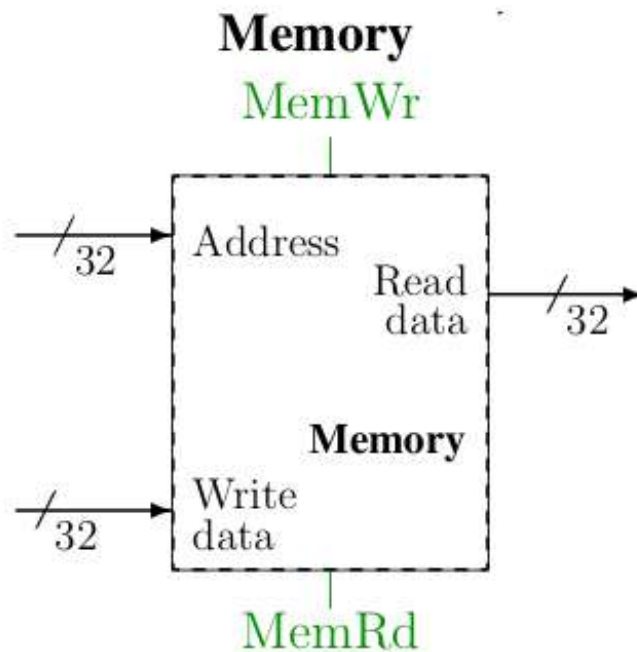
Por otro lado, utilizaremos registros de 32-bits con el objetivo de resolver tareas específicas, como almacenar el PC, el MBR, las entradas y salidas de la ALU, etc.



También contamos con un banco de 32 registros de 32 bits de propósito general, que permitirán la realización de una lectura o dos lecturas simultáneas. Esto se debe a que la mayoría de las instrucciones requieren de dos registros para operar, y si tuviésemos un banco de registros que solo nos pueda entregar un valor a la vez, eso nos multiplicaría las microinstrucciones de nuestro ciclo de instrucción. Como tenemos 32 registros, nos alcanza con 5 bits para poder identificarlos de manera unívoca.



Además, contamos con una única unidad de memoria, donde se encuentran almacenados tanto los programas como los datos sobre los que ejecutan.

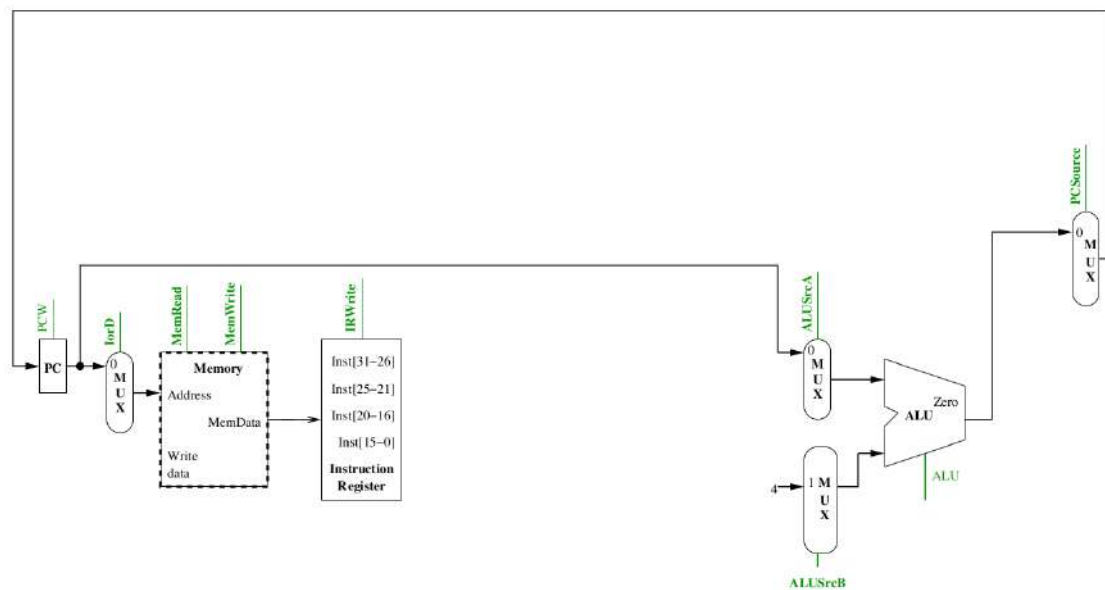


1.4.3. Paso 3: Construcción del camino de datos según los requerimientos con las componentes seleccionadas

Ahora tenemos que construir el camino de datos. Por un lado, ya seleccionamos las componentes necesarias, y sabemos construirlas con lo que vimos en lógica digital. Y por otro lado, tenemos nuestro set de instrucciones, que ya tenemos identificado, en un orden específico, dividido en sus diferentes etapas, cómo se van a ejecutar esas instrucciones a partir de mover datos de un lugar a otro.

Lo primero que tenemos que hacer es conectar las componentes para poder hacer el **Fetch**.

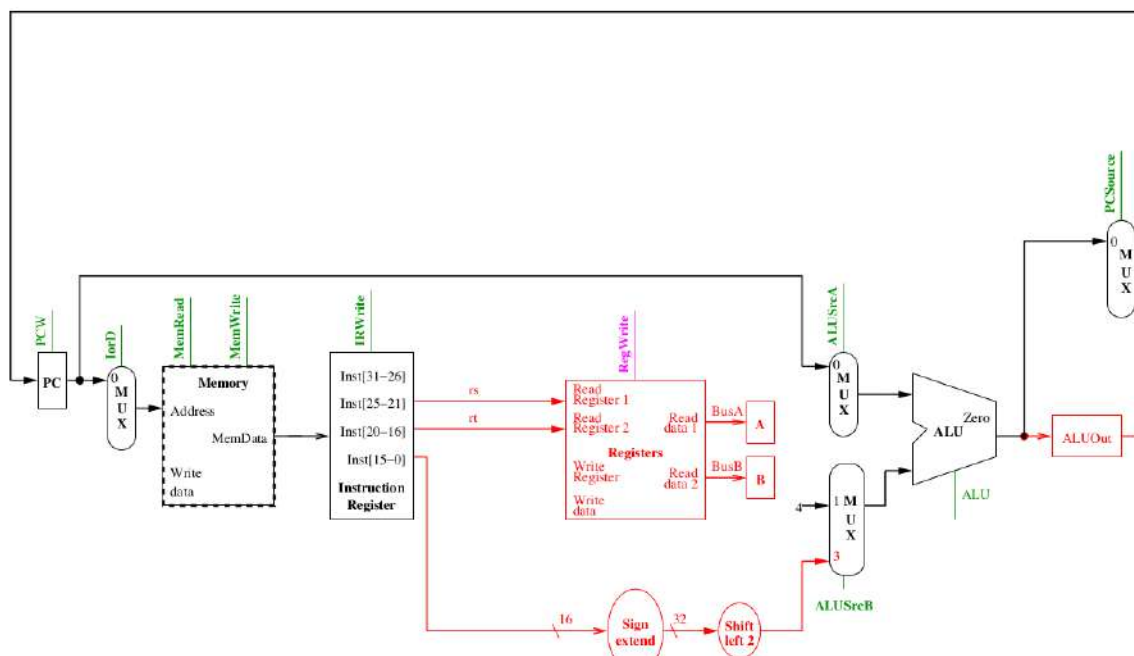
Instruction Fetch



Para poder hacer el Fetch, teníamos nuestro PC, con el cual accedíamos a la memoria. Tenemos un multiplexor entre el PC para que la memoria pueda ser accedida para obtener una instrucción (PC) o un dato, y se controla mediante la señal IorD (Instrucción o Dato). Esto también es necesario en la ALU para ambas entradas A y B.

Ahora necesitamos construir la lógica para poder ejecutar el **Decode** y lectura de registros.

Instruction Decode

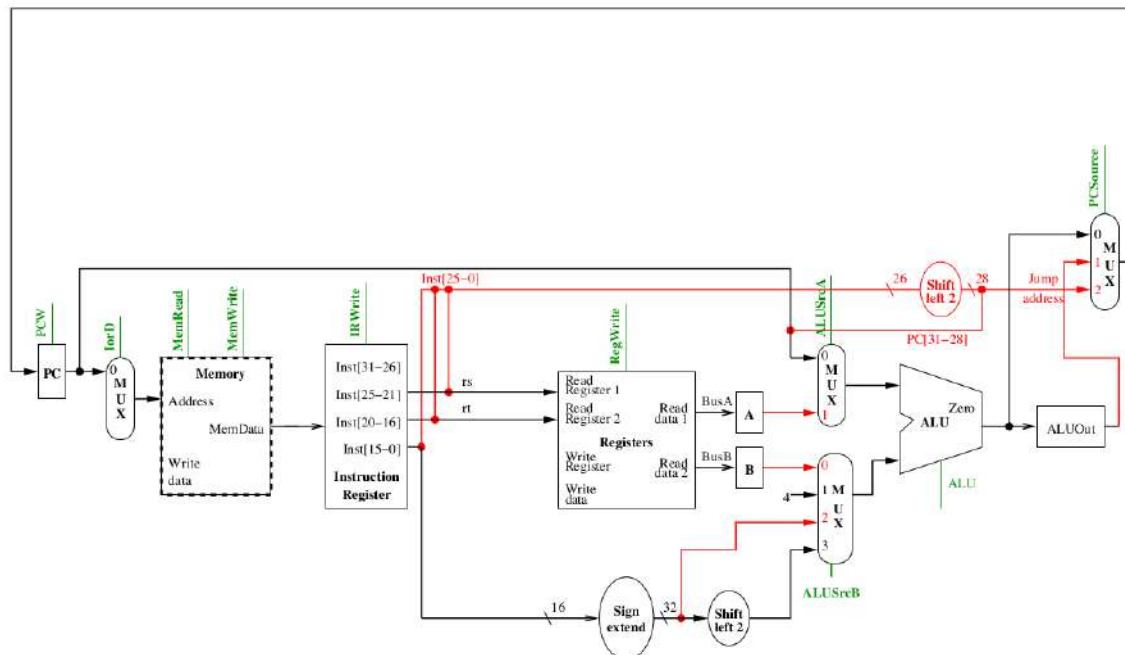


Tenemos un banco de registros, el cual va a ser indexado con bits que provienen del IR (ver modos de direccionamiento). Por este motivo, conectamos los bits correspondientes del IR al banco de registros, y la salida del banco de registro va a parar a registro específicos del banco (A y B). Además, para las instrucciones de Load y Store necesitamos poder extender el signo y shiftear los 16 bits del IR correspondientes al inmediato.

En la etapa de **Execute** tenemos que poder hacer varias cosas:

1. **ADDU, SUBU**: Sumar y restar el contenido de A y B.
2. **LW, SW**: Sumar el contenido de A con el inmediato (con signo extendido).
3. **BEQ**: Comparar lo que tengo en A y B, y si son iguales guardar en PC lo que tengo en ALUOut.
4. **J**: Actualizar el PC con pegar los primeros cuatro bits del PC y los 28 bits restantes del inmediato (shiftado en dos bits).

Instruction Execute

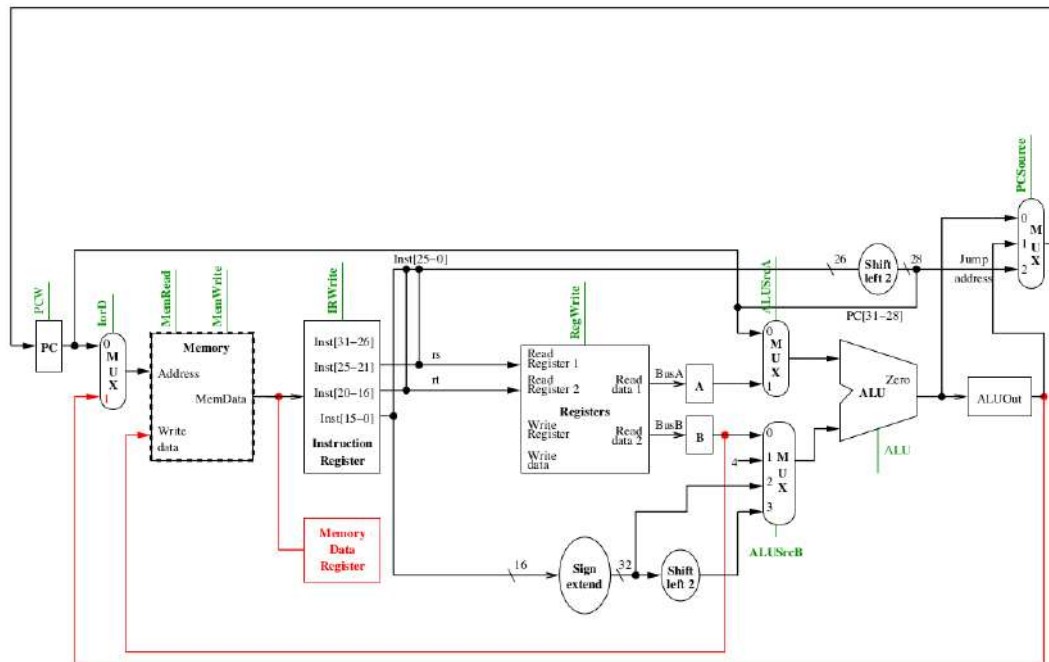


1. Para poder operar con A y B, los conectamos al multiplexor que se conecta a la ALU.
2. Conectamos los 16-bits del IR que corresponden al inmediato a el circuito que extiende con signo a 32-bits, y luego lo conectamos al multiplexor que se conecta a la ALU.
3. Conectamos ALUOut al multiplexor que se conecta con el PC (notemos que también habíamos conectado la salida Zero de la ALU a este multiplexor, implementando el if zero then ...). También agregamos un shifter, que usamos para shifear 2 bits del inmediato extendido con signo a 32 bits.
4. Juntamos los 26 bits del IR que usamos para la instrucción de salto, y luego lo conectamos a un shifter, para conectarlo al multiplexor que se conecta al PC.

En la etapa de **Memoria**, tengo que poder:

1. Traer de la memoria lo que tengo apuntado por ALUOut.
2. Guardar en la memoria lo que tengo en B.

Mem

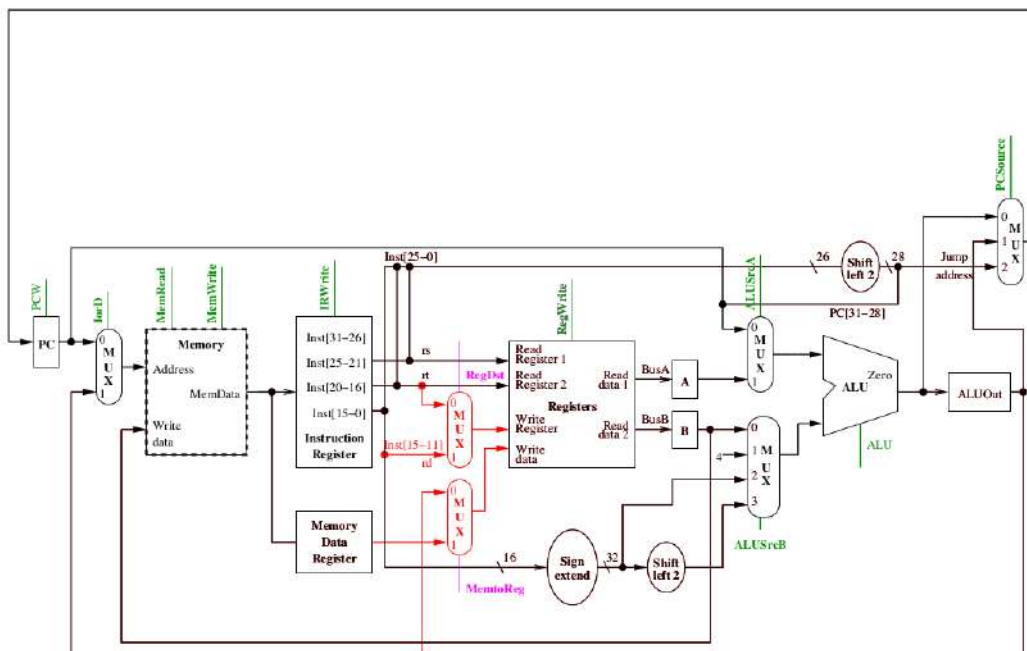


Para conseguir esto:

1. Conectamos ALUOut al multiplexor que se conecta a la entrada de direcciones de la memoria.
2. Conectamos B a la entrada de escritura de datos de la memoria.

Por último, en la etapa de **Write-Back** tengo que poder actualizar el banco de registros con:

1. La salida de la ALU.
2. El contenido del MDR.



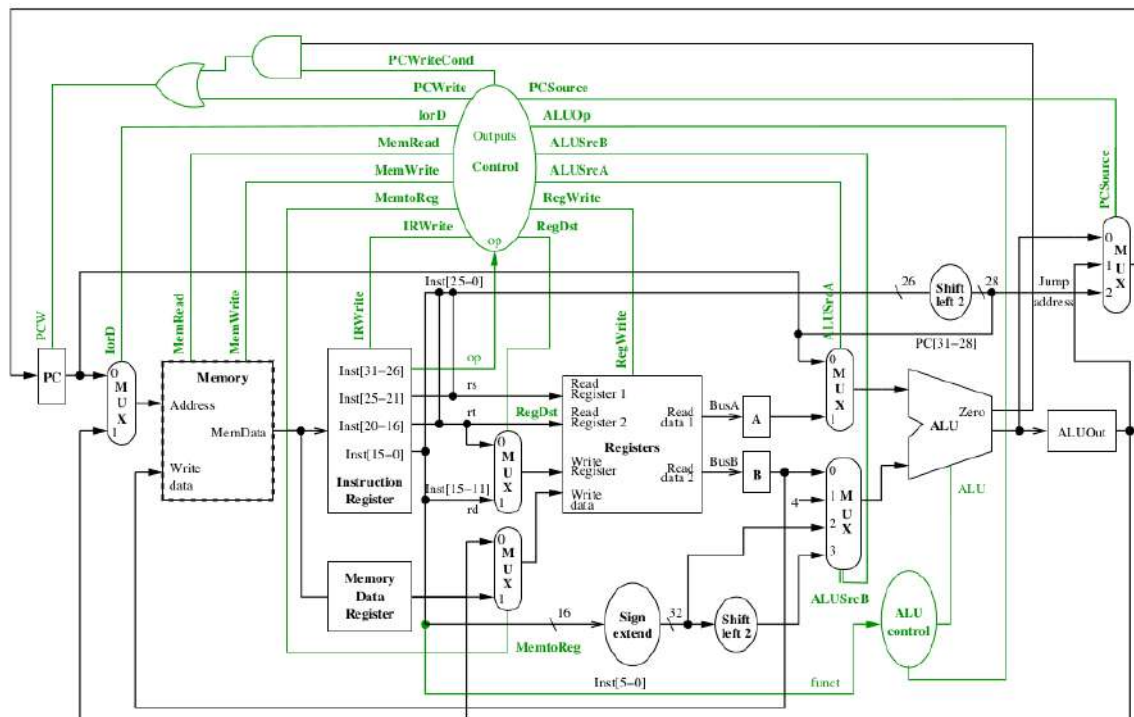
Para esto conectamos a ALUOut y al MDR al multiplexor que está en la entrada de escritura del

banco de registros.

1.4.4. Paso 4: Analizar la implementación de cada instrucción para determinar las señales de control necesarias

Nos falta construir el control, el cual decide qué parte del circuito está activa, en cada momento. Entonces, vamos a construir la unidad de control que me permite ejecutar paso a paso estos programas descritos en RTL, habilitando y deshabilitando determinadas secciones del camino de datos, para que la información pueda fluir de un lugar a otro.

Escencialmente, lo que vamos a hacer es que, por **turnos**, habilitar determinados lugar de esta circuitería, y deshabilitar el resto, dependiendo de la instrucción a ejecutar y de la etapa de ejecución de la instrucción.



Ahora, lo que tenemos que hacer es analizar las señales de control, que nos permiten decidir qué pasa en qué momento, y así poder determinar el valor que tienen que tener en cada momento, para obtener el resultado esperado.

Para esto, nos armamos una tabla que nos determina qué valores queremos que tengan las señales para que se realice determinada acción. Es decir, estamos asociando el valor de las señales a la acción que queremos que se realice:

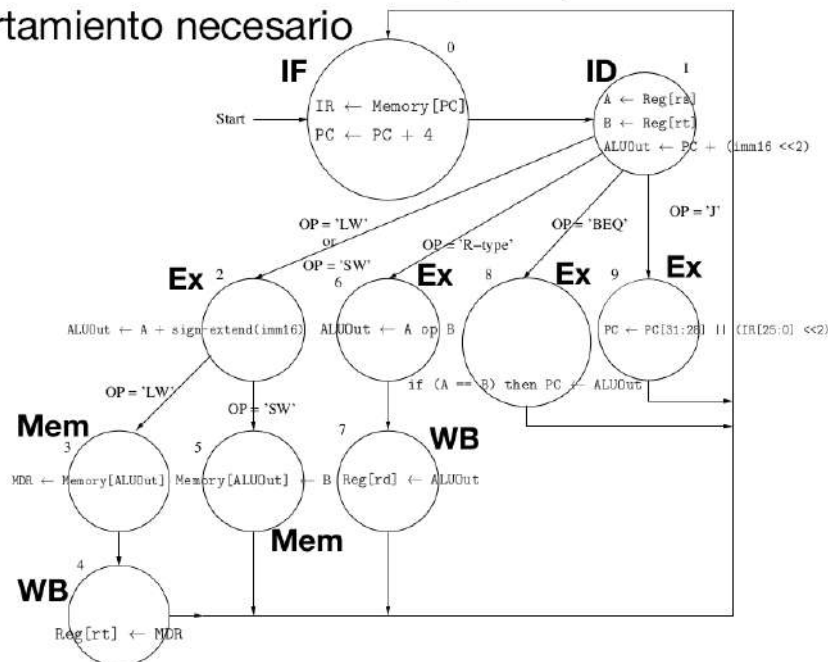
Señal	Acción	
	0	1
RegDst	El registro destino es rt	El registro destino es rd
RegWrite	No se escribe en el banco de registros	Si se escribe en el banco de registros
ALUSrcA	El primer operando de la ALU es el PC	El primer operando de la ALU es el registro A
MemRead	No se lee la memoria	El contenido de la memoria en la dirección especificada es colocado en el bus de datos
MemWrite	No se escribe la memoria	El contenido del registro B es escrito en la memoria en la dirección especificada
MemtoReg	El valor escrito en el banco de registros proviene de ALUOut	El valor escrito en el banco de registros proviene de MDR
lorD	La dirección proviene del PC	La dirección proviene del ALUOut
IRWrite	No se escribirá en el IR	Se escribirá en IR
PCWrite	---	Se escribe en el PC el valor de salida del MUX controlado por la señal PCSource
PCWriteCond	Si junto con PCWrite , no se escribe en PC	Se escribe el PC si el flag zero de la ALU está en 1

Señal	Valor	Acción
ALUOp	00	La ALU realizará una suma
	01	La ALU realizará una resta
	10	La ALU realizará la operación declarada en funct
ALUSrcB	00	El segundo operando de la ALU es el registro B
	01	El segundo operando de la ALU es 4
	10	El segundo operando de la ALU es la extensión con signo de los 16 bits menos significativos del IR
	11	El segundo operando de la ALU es la extensión con signo de los 16 bits menos significativos del IR decaído dos bits
PCSource	00	El PC se actualiza con PC + 4
	01	El PC se actualiza con el valor de ALUOut (el destino la operación BEQ)
	10	El PC se actualiza con el destino de la operación J

1.4.5. Paso 5: Construir la unidad de control que implemente el comportamiento necesario

Ya fabricamos la electrónica, y sistematizamos el significado de las señales de control, pero nos falta sistematizar un elemento adicional, el cual es el hecho de que hay que ejecutar las cosas por turnos (primero el Fetch, dependiendo de la instrucción que trajimos, un Decode u otro, etc). Entonces, tenemos que ejecutar paso a paso cada una de las etapas que forman a las instrucciones. Para esto, podemos representar la secuencia de pasos a ejecutar con el siguiente gráfico:

5. Construir la **unidad de control** que implemente el comportamiento necesario



Notemos que acá se ve la importancia de tratar de no separar cosas que pueden estar pegadas. En la etapa de Decode, podíamos leer un registro, dos registros, o leer dos registros y actualizar el ALUOut. Como estas no entran en conflicto, podemos realizar las 3, y tener un único caso para la etapa de Decode. Si no hubiéramos juntado estas tres acciones, el árbol se habría abierto en tres estados diferentes.

Entonces, ahora tenemos numerados los estados del 0 al 9. En cada estado, tenemos que realizar ciertas acciones (mover valores de un registro a otro, o a través de las componentes). Ahora nos queda asociar a cada estado los valores de que la señales de control tienen que tener para que se realice la acción deseada. Para esto, nos armamos la siguiente tabla:

	0	1	2	3	4	5	6	7	8	9
RegWrite	0	0	0	0	1	0	0	1	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemRead	1			1						
MemWrite	0	0	0	0	0	1	0	0	0	0
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWCond									1	
PCSource	00								01	10
ALUsrcA	0	0	1	1	1	1	1	1	1	
ALUsrcB	01	11	10	10	10	10	00	00	00	00
MemToReg					1			0		
RegDst					0			1		
IoD	0			1	1	1				
ALUOp	00	00	00				10		01	

Esta tabla me dice, en cada estado de mi ciclo de instrucción, qué señales de control tienen que estar en 0 y qué señales en 1. Con esto, podemos traducir el diagrama anterior del lenguaje RTL a los valores que tienen que tomar las distintas señales de control:

5. Construir la **unidad de control** que implemente el comportamiento necesario

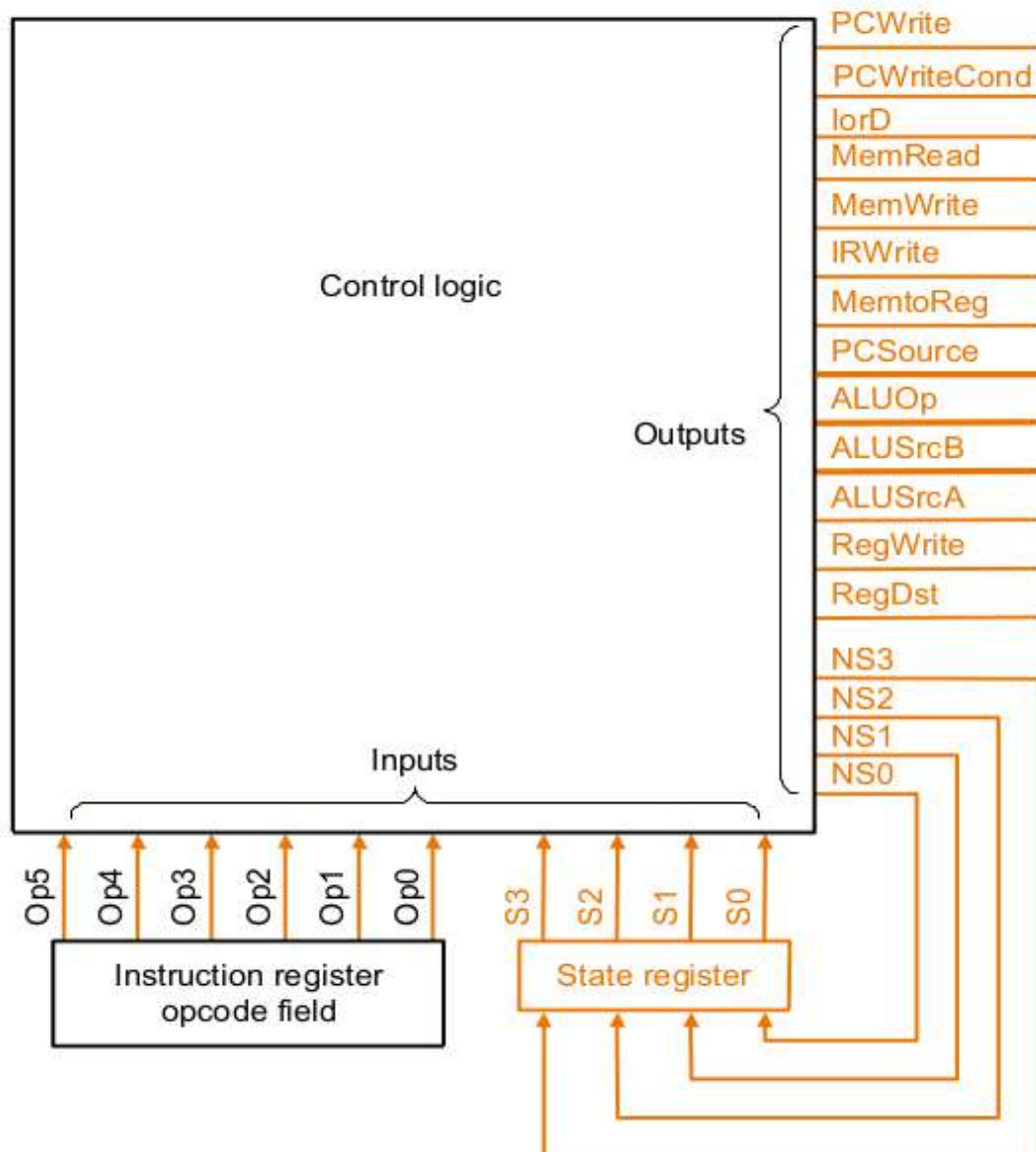


En resumen, hasta ahora lo que hicimos fue:

1. Identificar las instrucciones en lenguaje RTL que se tienen que ejecutar
2. Construir el camino de datos para poder ejecutar esas instrucciones, lo que nos determinó las señales de control.
3. Transformar la tabla que tenía sistematizada los programas en RTL a una máquina de estados (que empieza por el estado 0, y va ejecutando dependiendo de la instrucción, para volver al estado 0).

Nos falta llevar esta máquina de estados a la circuitería. Si vemos nuestro diagrama, notamos que las acciones que tenemos que hacer dependen del código de operación (6 bits) y del estado de la máquina.

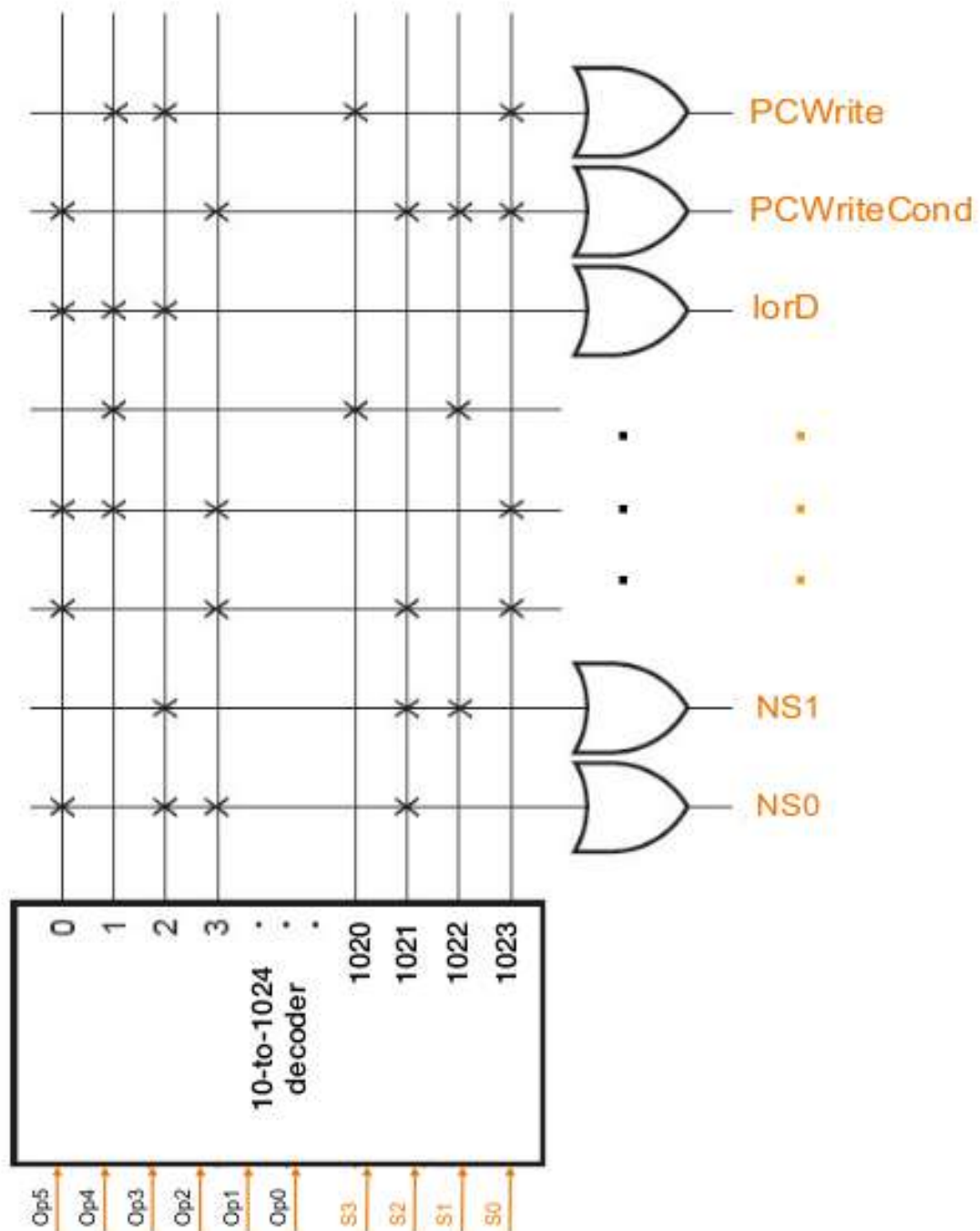
Para codificar el estado de la máquina, como solo tenemos 10 estados, nos alcanza con un registro de 4-bits (State Register). Este estado lo vamos a tener que avanzar como muestra el diagrama. Notemos que cuando llegamos al último estado de la instrucción, volvemos al estado 0.



1.5. Implementaciones de la Unidad de Control:

1.5.1. ROM

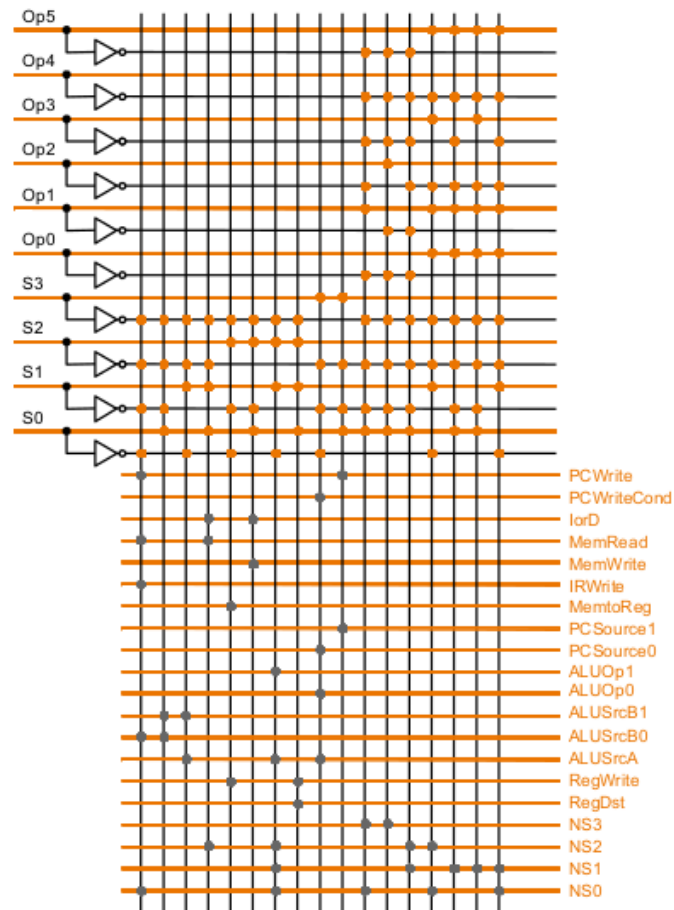
Lo último que nos falta hacer es construir la lógica de control para que, dado un opcode y un estado, la unidad de control envíe las señales de control correspondientes. La manera más sencilla de implementar esta lógica de control es utilizar una memoria ROM que tome direcciones de 10 bits (6 del opcode + 4 del State register), y devuelva la combinación correspondiente de señales:



Estas memorias son muy eficientes, funcionan electrónicamente, no tienen que guardar nada, se programan quemando fusibles, son muy rápidas. Sin embargo, como no todas las combinaciones de *opcode* + *estado* son posibles, hay mucho desperdicio.

1.5.2. PLA

Para resolver este problema, hay otro tipo de circuitos llamados PLA (Programmable Logical Array), que sirven para implementar fórmulas booleanas en forma normal disyuntiva:



Los PLA funcionan en dos etapas, primero se realizan los AND (20 x 17), y luego los OR (10 x 17)

ROM	PLA
10 bits de entrada	10 bits de entrada
20 bits de salida	17 bits de salida
1024 palabras de 20 bits	$10 \cdot 17 + 20 \cdot 17 = 460$ PLA cells
$1024 \cdot 20 = 20$ Kbits ROM	
La mayoría de las combinaciones no son utilizadas	17 combinaciones se utilizan

Cuadro 1.1: Comparación entre PLA y ROM

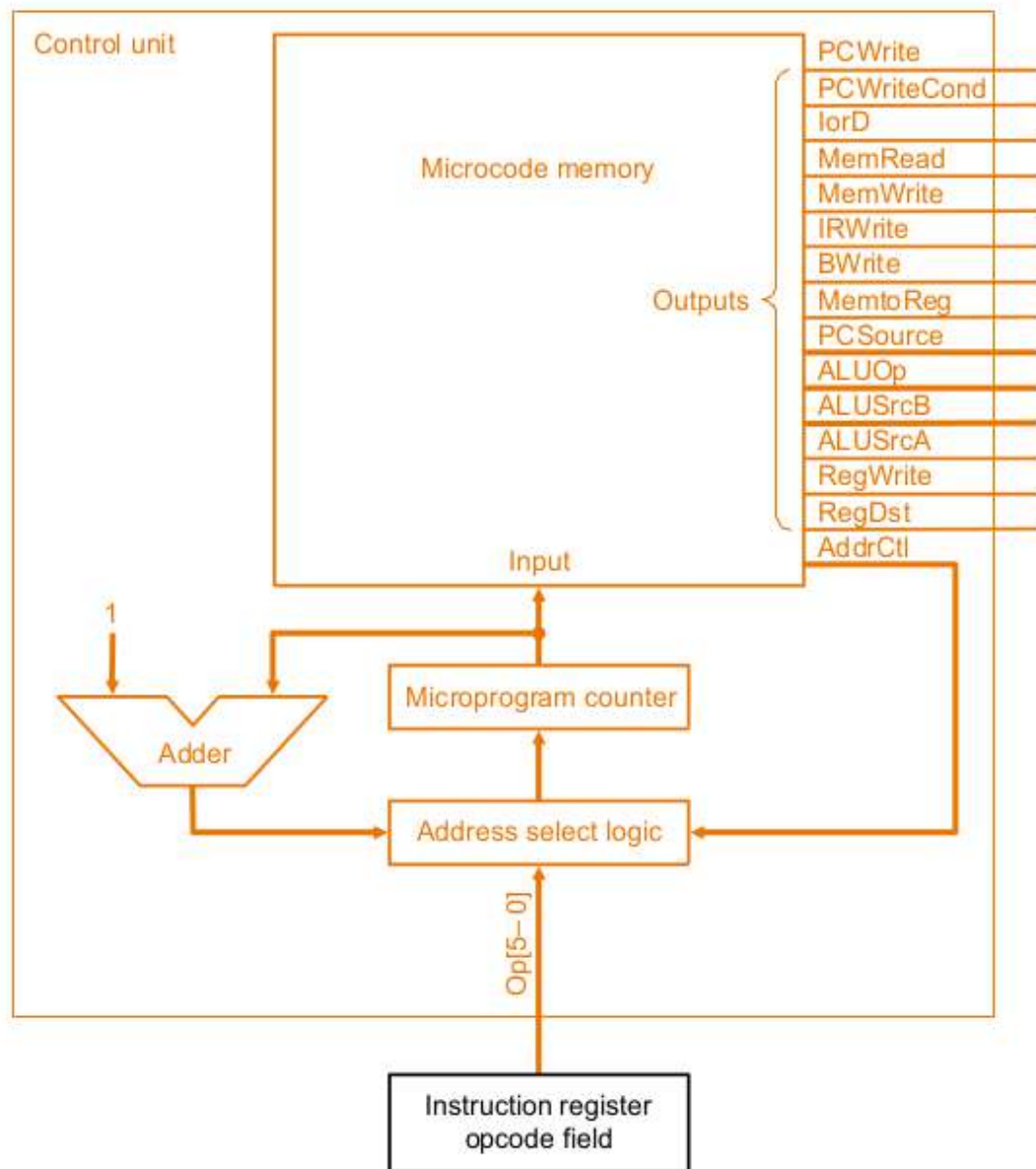
Esto resulta en un circuito más económico, pero al utilizar un muchas más compuertas lógicas, son circuitos más lentos que las ROMs.

1.6. Microprogramación

El diseño de una unidad de control debe incluir lógica para secuenciar a través de **microoperaciones**, para interpretar códigos de operación y para tomar decisiones basadas en los flags de la ALU, lo cual se puede volver un circuitaría demasiado compleja, y por lo tanto propensa a errores de implementación. Además de este diseño es relativamente inflexible, al tener que cambiar la lógica interna a nivel HW si quisiéramos agregar una nueva instrucción

Una alternativa, que se ha utilizado en muchos procesadores CISC (con cientos de instrucciones y modos), es implementar una unidad de control microprogramada. La idea es simplificar la lógica de control a partir de reducir instrucciones complejas a una secuencia de **microinstrucciones** (guardadas en una memoria de uso específico), que son más fáciles de manejar.

Recordemos que a alto nivel teníamos una ISA con instrucciones que se traducían a señales que nos permitían configurar nuestra unidad de control. Ahora, lo que vamos a hacer es pensar que la ejecución, siguiendo las etapas del ciclo de instrucción, van a ser micro-instrucciones específicas para esa máquina, mucho más sencillas.



De esta manera, podemos formar palabras de control para representar secuencias de micro-operaciones realizadas por la unidad de control, a las que llamaremos **microinstrucciones**. Luego, colocamos estas microinstrucciones en una memoria, asignándole a cada palabra una dirección única.

Ahora agregamos un campo de dirección a cada palabra de control, indicando la ubicación de la siguiente palabra que se ejecutará, si alguna condición fuera verdadera (por ejemplo, el bit indirecto en una instrucción es 1). Además, agregamos algunos bits para especificar dicha condición.

Estas microinstrucciones pueden ser organizadas en una **memoria de control** (típicamente una ROM). El conjunto de microinstrucciones que conforma a cada instrucción debe ejecutarse de forma secuencial (aumentando el microPC). Cada secuencia de microinstrucciones termina con una microinstrucción de salto o branch que indica a dónde ir a continuación (al inicio de la próxima secuencia de microinstrucciones que conforman la siguiente instrucción).

En el microPC nos guardamos la dirección de la próxima microinstrucción a ejecutar.

1.7. Ventajas y Desventajas

La principal ventaja del uso de la microprogramación para implementar una unidad de control es que simplifica el diseño de la unidad de control. Por lo tanto, es más barato y menos propenso a cometer errores en la implementación.


La principal desventaja de una unidad microprogramada es que será algo más lenta que una unidad cableada de tecnología comparable. A pesar de esto, la microprogramación es la técnica dominante para implementar unidades de control en arquitecturas CISC puras, debido a su facilidad de implementación. Los procesadores RISC, con su formato de instrucción más simple, suelen utilizar unidades de control cableadas.

Capítulo 2

Entrada / Salida

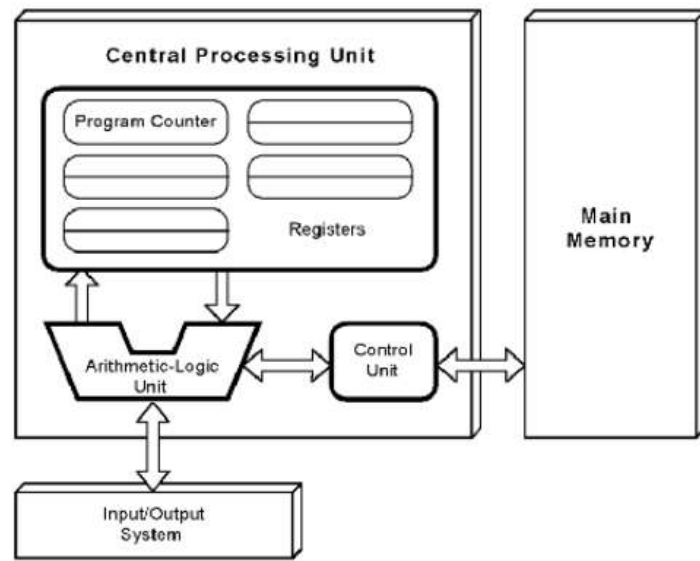
2.1. Introducción

En general, vamos a estudiar los eventos necesarios para el manejo de entrada / salida que se sitúan a nivel 0, salvo algunos que ocurren a nivel 2. Es decir, cuando hablamos del manejo de dispositivos de entrada / salida, tenemos componentes electrónicos (nivel 0), pero también hay componentes de software (los dispositivos van a ser manipulados a través de programas).



Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

En el modelo de computo Von Neumann - Turing tenemos una cinta infinita en la que se almacenan los programas, los datos, y los resultados. Necesitamos algún mecanismo para hacer input / output a esta memoria, que permita cargar los datos y programas, y también recuperar los resultados.



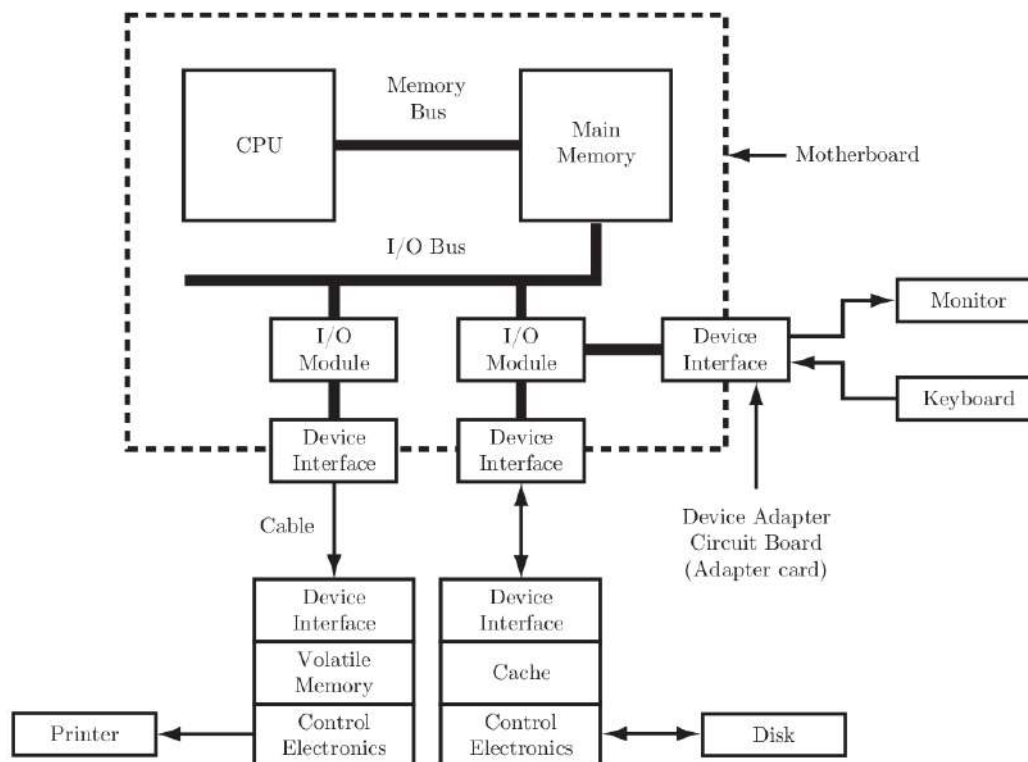
Hasta ahora, estuvimos asumiendo que los datos ya se encontraban en la memoria. Ahora necesitamos construir el tercer componente importante de nuestra máquina, que son los mecanismos por los cuales los datos llegan y salen de la memoria.

Este tercer elemento es el Subsistema de E/S, el cual se encarga de conectar nuestros elementos que utilizamos para computar (CPU, memoria) con el mundo exterior (de donde provienen los datos), a partir de dispositivos conectados de alguna manera al equipo.

Una computadora no sirve de nada sin algún medio para introducir datos y para enviar información. La mayoría de las veces en las que una computadora funciona lenta, no se debe al procesador o la memoria, sino en cómo el sistema procesa su entrada y salida (E/S).

Definiremos al sistema de E/S como el conjunto de componentes que mueven datos entre dispositivos externos y la CPU o la memoria principal. Los sistemas de E/S incluyen, pero no se limitan a:

- Espacios de memoria principal dedicados a funciones de E/S
- Buses que permiten mover datos dentro y fuera del sistema
- Módulos de control de computadora y de dispositivos periféricos
- Interfaces a componentes externos (como teclados y discos)



Los módulos de E/S se encargan de mover datos entre la memoria principal y una interfaz de dispositivo en particular. Las interfaces están diseñadas específicamente para comunicarse con ciertos tipos de dispositivos, como teclados, discos o impresoras. Las interfaces manejan los detalles para asegurarse de que los dispositivos estén listos para el siguiente bloque de datos, o que la computadora esté lista para recibir el próximo bloque de datos provenientes del dispositivo periférico.

2.2. Métodos de acceso a E/S

Cada controlador, de cada dispositivo de E/S, tiene un par de registros que son usados para comunicarse con la CPU. Escribiendo en esos registros, el SO puede enviar comandos al dispositivo para que realice alguna acción (transferir datos, apagarse, prenderse, etc). Leyendo de esos registros, el SO puede saber cuál es el estado del dispositivo, si es que está preparado para aceptar un nuevo comando, etc.

Además de los registros de control, muchos dispositivos tienen un buffer sobre el que el SO puede leer y escribir. Por ejemplo, una manera para que las computadoras puedan mostrar píxeles por pantalla es tener una RAM de video, que consiste en un buffer de datos que está a disposición de los programas y el SO para que puedan escribir sobre el mismo.

Ahora, ¿cómo hace la CPU para comunicarse con los registros de control y los device data buffers?. Existen dos alternativas, que pueden ser usadas en combinación: **Puertos de E/S** y **Mapeo a Memoria**.

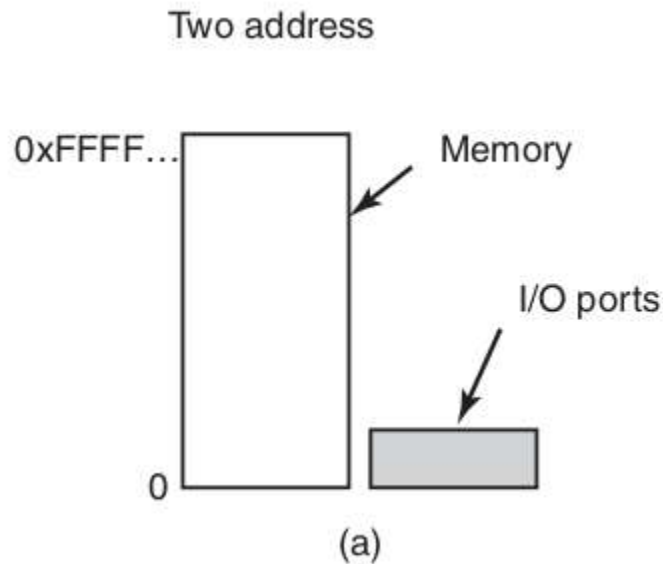
2.2.1. Puertos dedicados

Para la primera alternativa, cada registro de control tiene asignado un **puerto de E/S**, un entero de 8 o 16 bits. El conjunto de todos los puertos de E/S forman el *I/O port space*, que es un espacio de E/S protegido para evitar que los programas de nivel usuario puedan acceder al mismo (solo el kernel puede), distinto al espacio de memoria de la memoria principal.

La CPU puede leer y escribir sobre los registros de control direccionados por PUERTO utilizando instrucciones especiales de E/S:

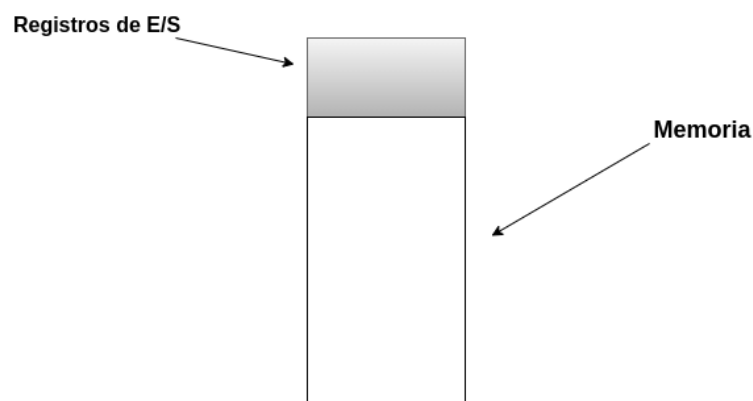
- **IN REG, PUERTO**
- **OUT PUERTO, REG**

En este esquema de acceso a dispositivos de E/S, los espacios de memoria y el espacio de E/S son distintos:



2.2.2. Mapear a Memoria

La segunda opción es *mapear* todos los registros de control en el espacio de memoria principal. Cada registro de control es asignado a una **única dirección de memoria**, que no puede ser utilizada por el resto del sistema. En general, las direcciones asignadas suelen estar en alguno de los extremos, es decir o bien en las direcciones bajas o bien en las altas.



2.2.3. Puertos vs Mapeo a Memoria

Ambos esquemas para direccionar a los controladores tienen distintas ventajas y desventajas.

Ventajas de usar Memory-Mapped I/O

En primer lugar, las instrucciones especiales de E/S (IN, OUT) son necesarias para el acceso (desde el SO) a los registros de control. Sin embargo, no existen estas instrucciones en los lenguajes de alto nivel, por lo que requieren del uso de código ensamblador, lo cual agrega un overhead al manejo de E/S. En cambio, usar MMap-I/O no requiere instrucciones especiales, por lo que todo el driver puede ser escrito en lenguaje C.

En segundo lugar, con MMap-I/O no se requieren mecanismos de protección para evitar que los procesos de nivel usuario realicen I/O. Todo lo que debe hacer el SO es evitar mapear el espacio de memoria dedicado a los registros de control en el espacio virtual de direcciones del usuario. Incluso, si cada dispositivo tiene su registro de control en una página distinta del espacio de direcciones, el SO puede dar control al usuario sobre dispositivos específicos, evitando que los drivers puedan interferir unos con otros.

Por último, con MMap-I/O cada instrucción que puede referenciar memoria, puede referenciar a los registros de control, y así evitar tener instrucciones específicas para trabajar sobre los registros de control.

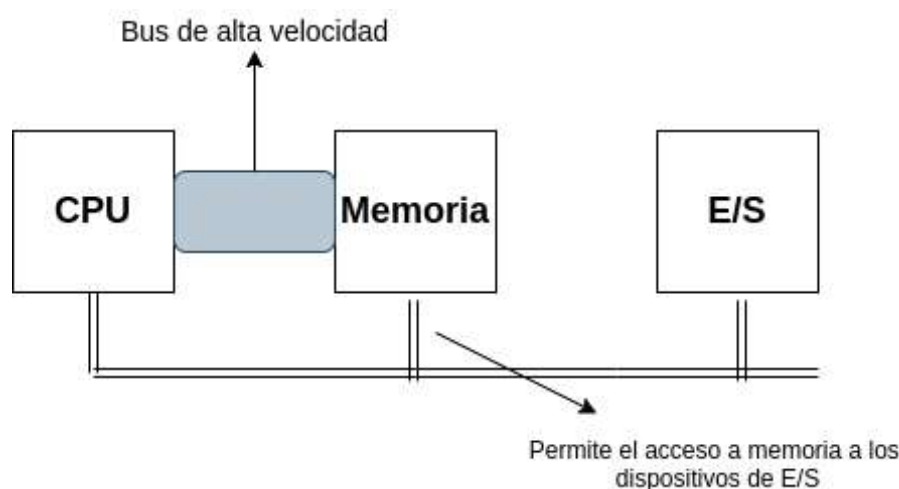
Ventajas de usar Puertos dedicados a E/S

En primer lugar, la mayoría de las computadoras utilizan alguna forma de memoria cache. Guardarse en una cache el registro de control resultaría en desastre, ya que las referencias a las direcciones asociadas a los registros de control resultarían en lecturas en la cache, y no preguntarían al dispositivo. Es decir, el dispositivo no tendría manera de comunicarse con el sistema.

Para evitar esta situación con MMap-I/O, el HW debe poder deshabilitar el caching de las páginas en las que se encuentren los registros de control. Esto añade una complejidad extra, que no es necesaria usando puertos dedicados a E/S.

En segundo lugar, si sólo hay un espacio de direcciones, entonces todos los módulos de memoria, y todos los dispositivos de E/S deben examinar todas las referencias a memoria para saber a cuáles deben responder.

Esto es sencillo en caso de tener un único bus de datos, pero en las computadoras modernas se tiene un bus dedicado de alta velocidad para acceder a la memoria, lo cual complica separar los accesos a la memoria, de los accesos a los registros de control.



El problema con tener buses de memoria en máquinas con MMap-IO es que los dispositivos de E/S

no tiene manera de ver las direcciones de memoria que viajan por el bus de alta velocidad, y por lo tanto no puede contestar a los pedidos.

Nuevamente, se requieren medidas especiales para que MMap-IO funcione en este tipo de máquinas. Una posibilidad es primero mandar todas las referencias de memoria por el bus de alta velocidad. Si no se recibe respuesta, la CPU intenta comunicarse con el resto de los buses.

Una segunda opción es poner un *snooping device* en el bus de alta velocidad, para que este dispositivo envíe todas las direcciones que potencialmente sean para dispositivos de E/S. Esto puede generar cuellos de botella, ya que los dispositivos de E/S no puede procesar a la misma velocidad que la memoria.

Un tercer diseño posible consiste en filtrar las direcciones en el controlador de la memoria. En este caso, el controlador de la memoria contiene registros de rango precargados al momento de booteo. Las direcciones que caen dentro de uno de los rangos marcados como nonmemory son enviados a los dispositivos de E/S, en lugar de la memoria. Este esquema requiere tener un mecanismo que permita decidir, en tiempo de booteo, qué direcciones son, efectivamente, de memoria, y cuales no.

2.3. Métodos de control de E/S

El otro punto central a la hora de analizar E/S es de qué manera se realiza la solicitud de atención de E/S.

Normalmente los dispositivos tienen un registro de estado, el cual cambia cuando se escribe sobre el registro de datos a not ready. Luego de terminar de procesar esos datos, este registro de estado se pone en ready. La pregunta es, ¿Cómo hacemos para saber cuándo el dispositivo se encuentra disponible para realizar algún trabajo?

Típicamente se utilizan cualquiera de los cuatro métodos generales de control de E/S. Estos métodos son **programmed I/O**, **interrupt-driven I/O**, **direct memory access**, y **channel-attached I/O**. Aunque un método no es necesariamente mejor que otro, la forma en que una computadora controla su E/S influye en gran medida en el diseño y el rendimiento general del sistema. Nuestro objetivo es saber cuándo el método de E/S empleado por una arquitectura de computadora en particular es apropiado para cómo el sistema se utilizará.

2.3.1. Programmed Input / Output: Polling

La manera más simple de E/S es hacer que la CPU haga todo el trabajo. Para esto, los sistemas que utilizan programmed I/O dedican al menos un registro para uso exclusivo de cada dispositivo de E/S. La CPU monitorea continuamente cada registro, esperando a que los datos lleguen. Este mecanismo se lo conoce como **polling**.

La CPU se queda constantemente revisando el estado del dispositivo, y una vez que la CPU detecta una condición de ready, actúa de acuerdo con instrucciones programadas para ese registro en particular.

El beneficio de utilizar este enfoque es que tenemos control programático sobre el comportamiento de cada dispositivo. Los cambios de programa pueden hacer ajustes al número y tipos de dispositivos en el sistema, así como a sus prioridades e intervalos de sondeo.

Sin embargo, el sondeo constante de registros es un problema. La CPU está en un bucle continuo de *busy waiting* hasta que comienza a atender una solicitud de E/S. Es decir, no hace ningún trabajo útil hasta que hay E/S para procesar. Debido a estas limitaciones, utilizar programmed I/O es un mecanismo más adecuado para sistemas de propósito específico, como cajeros automáticos y sistemas que controlan o monitorean eventos ambientales, pero para sistemas más complejos necesitamos un mejor método de manejo de E/S.

En resumen, cuando hacemos PIO:

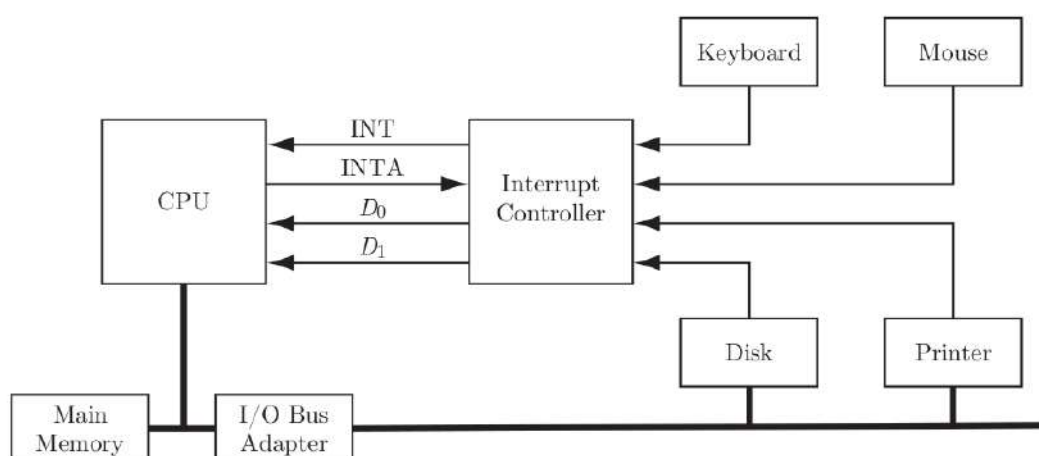
- El driver periódicamente verifica si el dispositivo se comunicó.
- Los cambios de contexto son controlados.
- Puede llegar a consumir mucha CPU en busy waiting.

2.3.2. Interrupciones

Las interrupciones pueden considerarse lo contrario de polling. En lugar de que la CPU pregunte continuamente a sus dispositivos conectados si tienen alguna entrada, los dispositivos le dicen a la CPU cuándo tienen datos para enviar. La CPU continúa con otras tareas hasta que un dispositivo solicita una interrupción. Las interrupciones generalmente se habilitan globalmente con un bit en el registro de flags de la CPU llamado *interrupt flag* (IF).

Una vez que se establece el IF, el sistema operativo interrumpe cualquier programa que se esté ejecutando actualmente, guardando el estado de ese programa y la información variable. Luego, el sistema obtiene el puntero al inicio de la rutina de atención de interrupciones (ISR) correspondiente, y lo carga en el PC. Cuando se termina de ejecutar esta rutina, se reestablece la información que se había guardado para regresar al programa que había sido interrumpido.

Esquemáticamente, el mecanismo de interrupciones funciona de la siguiente manera:



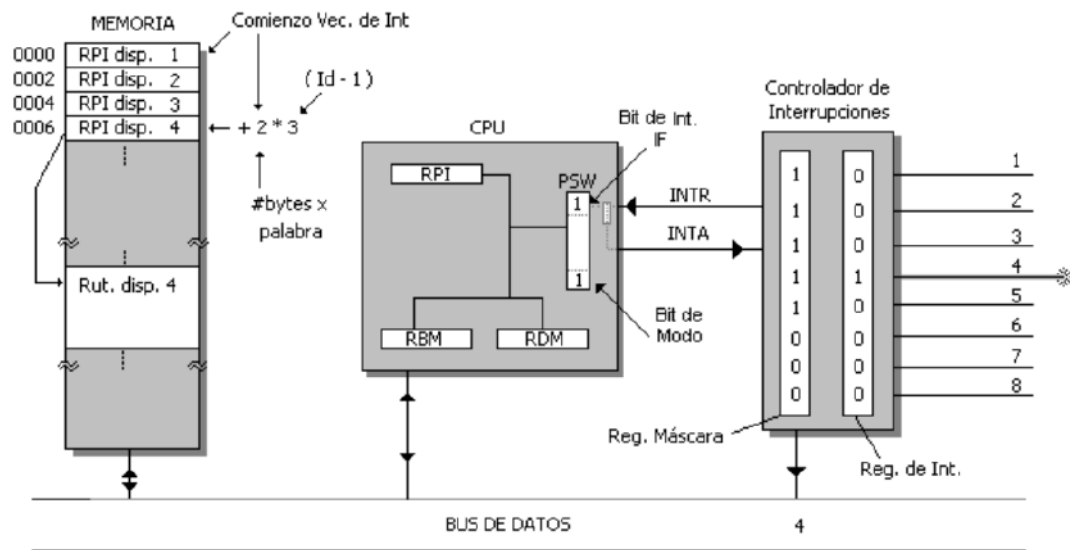
Tenemos una serie de dispositivos (teclado, mouse, disco, impresora) conectado a un **controlador de interrupciones**, el cual tiene una señal para informar a la CPU que quiere generar una interrupción (INT), y la CPU tiene una señal para notificar al controlador de interrupciones que, efectivamente, puede ser interrumpida (INTA).

Una vez que el controlador de interrupciones recibe el INTA (INTerrupt Acknowledge), este avisa, a partir de ciertas señales que viajan por el bus de datos (D_0 , D_1), a la CPU cuál dispositivo solicitó la interrupción (tenemos 4 dispositivos, con dos bits nos alcanza para identificarlos de manera unívoca).

Luego, la comunicación se realiza directamente entre la CPU y el dispositivo, sin la intervención del controlador de interrupciones. En resumen, el controlador de interrupciones se encarga de manejar las solicitudes de interrupción, mientras que la CPU se encarga de atenderlas (ejecutando la ISR correspondiente).

Notemos que este intercambio requiere de hardware específico para el manejo de señales físicas (nivel 0) para la solicitud de interrupciones, y al mismo tiempo, la CPU tiene que tener la capacidad de atender las interrupciones, a partir de instrucciones específicas (nivel 2).

Al momento de solicitar las interrupciones, necesitamos de algún mecanismo que nos permita jerarquizar los pedidos, para evitar que la máquina sea constantemente interrumpida. Veamos cómo es este mecanismo en el 8086:



Tenemos un controlador de interrupciones, el cual contiene un registro de interrupciones, que guarda las señales de solicitud de los distintos dispositivos (para solicitar interrupción se coloca un 1), y un registro máscara, el cual se utiliza para habilitar o deshabilitar las solicitudes de los dispositivos, en base a la jerarquía de los mismos (1 indica que la interrupción está permitida).

Cuando llega una interrupción que está habilitada (dispositivo 4 en la imagen), el controlador de interrupciones le avisa a la CPU, mediante una señal (INTR), que llegó este pedido.

Las interrupciones solo pueden ser atendidas únicamente al final del ciclo de instrucción. En el momento en el que se termine este ciclo, la CPU revisa si las interrupciones están habilitadas globalmente ($IF = 1$, del PSW), y que la señal INTR está alta.

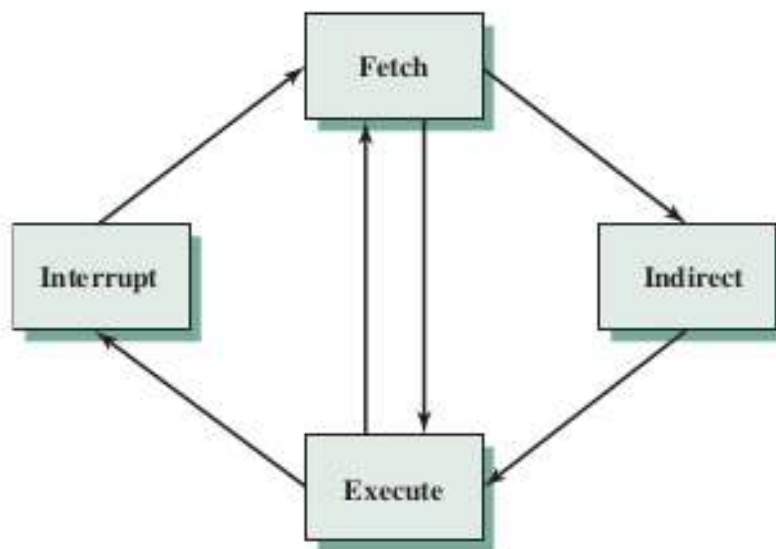


Figure 14.4 The Instruction Cycle

Si lo están, notifica al controlador de interrupciones que se encuentra disponible para atender la interrupción, levantando el INTA.

Una vez que el controlador de interrupciones recibe el INTA, notifica, a través del bus de datos, el número de dispositivo que solicitó la interrupción, para luego poder encontrar en qué dirección se encuentra la ISR.

El número recibido por el bus de direcciones es usado para indexar en una tabla llamada **interrupt vector**, donde podemos encontrar los punteros que apuntan al inicio de las ISRs (Interrupt Service Routine). La ubicación del interrupt vector puede ser cableada dentro de la máquina, o puede estar en algún lugar de la memoria, con un registro de la CPU apuntando a su origen.

Antes de colocar este puntero en el PC, la CPU guarda el contexto de ejecución del programa en la pila (PSW y PC en nuestro caso). Esto se hace para poder volver a la ejecución del programa interrumpido, una vez se termine de ejecutar la ISR.

Mecanismo de interrupción 8086

En resumen, el mecanismo de interrupción es el siguiente:

A nivel HW:

1. El controlador del dispositivo de I/O activa la señal de interrupción a la que se encuentra conectado.
2. La CPU termina de ejecutar la instrucción en curso y verifica si el flag IF está en 1 y la señal INTR está alta.
3. En este caso, sube la señal INTA.
4. El PIC envía por el bus de datos el identificador del dispositivo que produjo la interrupción.

-
5. Guarda el contexto del programa en la pila (PSW y PC).
 6. Deshabilita las interrupciones globalmente usando el flag IF colocándolo en 0.
 7. Se pasa el procesador a modo kernel para que la rutina se ejecute con privilegios de sistema operativo.
 8. Se carga en el PC el puntero que apunta a la ISR.

A nivel SW:

1. Se guarda la máscara de interrupciones.
2. Se modifica la máscara a fin de habilitar selectivamente las interrupciones que permitiremos durante la ejecución de la rutina.
3. Se habilitan globalmente las interrupciones colocando el flag IF en 1.
4. Se ejecuta la rutina de atención de la interrupción solicitada.
5. Se deshabilitan globalmente las interrupciones colocando el flag IF en 0.
6. Se reemplaza la máscara de interrupciones con la original.
7. Retorna de la interrupción con una instrucción que restaura el estado del programa antes del llamado (IRET)
 - a) Restaura el PC de la pila.
 - b) Restaura el PSW de la pila (datos de interrupción).

En resumen:

- El dispositivo avisa generando una interrupción a la CPU.
- Sirve para eventos asíncronos poco frecuentes.
- Los cambios de contexto son impredecibles.

2.3.3. DMA

Una crítica que se le hace a las interrupciones es que realizar los cambios de contexto lleva tiempo, y si estamos realizando constantemente interrupciones (cuando manejamos grandes cantidades de información), se pierde una gran cantidad de tiempo de CPU.

Una posible solución es utilizar DMA. La idea de DMA es aprovechar que los trabajos que tiene que hacer la CPU para manejo de E/S son lo suficientemente fáciles como para estar programados en un chip dedicado (controlador de DMA) mucho más sencillo que la CPU.

Luego, la idea es hacer polling, pero haciendo que el controlador de DMA haga todo el trabajo, aislando la comunicación entre la memoria y los dispositivos de la CPU, permitiendo que la CPU continúe ejecutando otro programa.

Para que el SO pueda usar DMA, se necesita tener un controlador de DMA. Este tiene acceso al bus de sistema, de manera independiente a la CPU. Normalmente, hay un único controlador de DMA disponible, que se encarga de regular las transferencias entre múltiples dispositivos, incluso de manera concurrente.

El controlador posee varios registros que pueden ser leídos y escritos por la CPU. Esto incluye a un memory address register, un byte counter register, y varios registros de control, que se encargan de especificar el puerto de E/S a utilizar, la dirección de la transferencia (lectura, o bien escritura sobre el dispositivo de E/S), y el número de bytes a transferir por ráfaga (burst).

Para poder explicar cómo DMA funciona, primero veamos cómo una lectura de disco ocurre sin utilizar DMA.

Primero, el controlador del disco lee el bloque (uno o varios sectores) del disco, de forma serial, bit por bit, hasta que el bloque completo se encuentre en el buffer interno del controlador. Luego, computa el checksum para verificar que no hubieron errores en la lectura.

Luego, el controlador genera una interrupción. Cuando el SO empieza a ejecutarse, puede leer el bloque del disco, desde el buffer/registro del controlador del dispositivo, un byte o palabra a la vez.

Ahora vemos qué pasa cuando se utiliza DMA, al ser el procedimiento distinto.

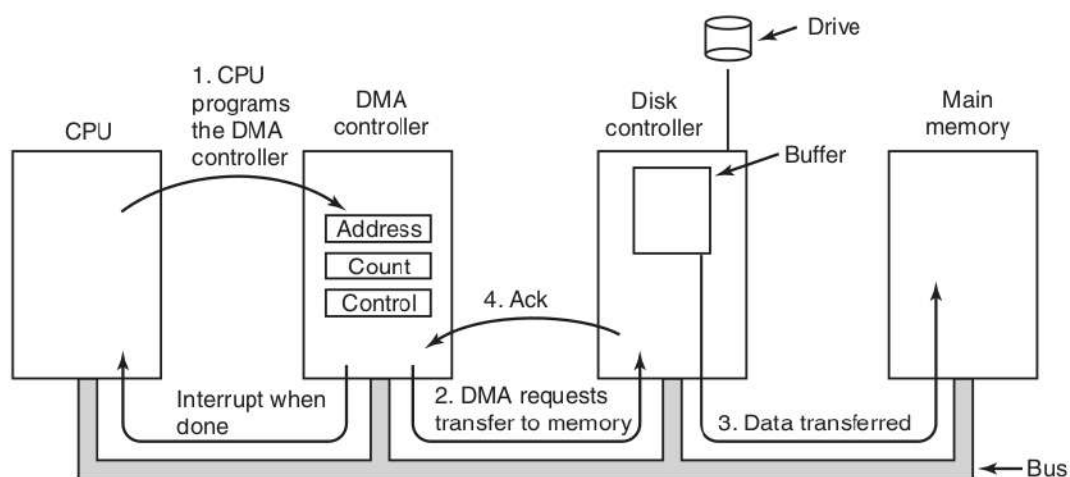


Figure 5-4. Operation of a DMA transfer.

PASO 1:

Primero la CPU programa al controlador de DMA, seteando los registros de control, de manera de que este sepa qué transferir y a dónde. También emite un comando al controlador del disco diciéndole que lea la data (y que se la guarde en el buffer), para que luego verifique el checksum. Una vez la data esté validada, el DMA puede iniciar.

PASO 2:

El controlador de DMA inicia la transferencia emitiendo un pedido de lectura por el bus, al controlador del disco. Este pedido de lectura es como cualquier otro, y al controlador del disco no le importa si fue el controlador de DMA o la CPU. Típicamente, la dirección de memoria a la cual el controlador de disco debe escribir es enviada por las líneas de direcciones del bus, de manera que cuando el controlador del disco obtenga la siguiente palabra, sepa donde debe escribir. Entonces, el controlador del disco escribe a las direcciones de memoria que le vaya indicando el controlador de DMA.

PASO 3:

Cuando la escritura es completada, el controlador del disco envía una señal de acknowledgement al controlador de DMA.

PASO 4:

Luego, el controlador de DMA incrementa la dirección de memoria a la cual escribir, y decrementa el registro con el byte count. Los pasos 2 - 4 son repetidos hasta que el count llegue a 0. En ese momento, el controlador de DMA interrumpe a la CPU para informarle que la transferencia fue completada.

Crítica a DMA:

No todas las computadoras usan DMA. La desventaja que puede tener es que la CPU es, en muchos casos, mucho más rápido que el controlador de DMA, y por lo tanto puede hacer el trabajo mucho más rápido (siempre que el factor limitante no sea la velocidad del dispositivo de E/S). Además, tener un el controlador de DMA es más costoso que tener a la CPU haciendo todo el trabajo, lo cual es importante cuando trabajamos con sistemas embebidos.

En resumen:

- Es útil para transferir grandes volúmenes de datos (la CPU no interviene).
- Requiere de un componente de HW particular (el controlador de DMA).
- Cuando el controlador finaliza la transferencia, interrumpe a la CPU.

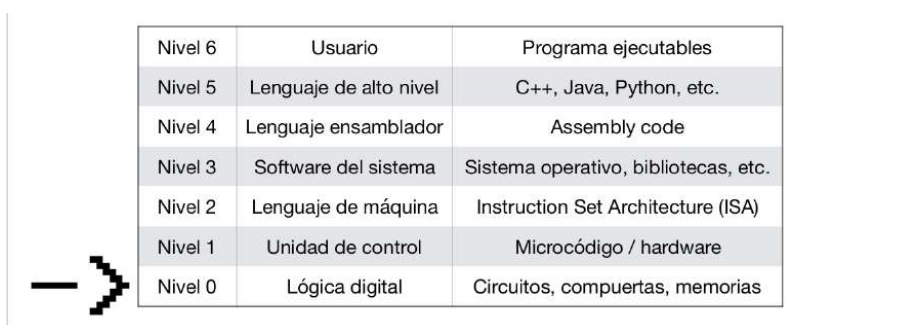
Capítulo 3

Entrada / Salida: Conversión de señales

3.1. Introducción

Vamos a estudiar cómo funciona la conversión de señales para comprender a mayor profundidad qué significa conectar la computadora al mundo exterior. Hasta ahora, vimos cómo manejar dispositivos e interrupciones. Sin embargo, todavía no vimos cómo podemos adquirir grandes volúmenes de información externas a nuestra máquina.

Uno de esos mecanismos es la captación de información del mundo real, digitalizarla y almacenarla como información digital. Este mecanismo se conoce como **conversión de señales**.



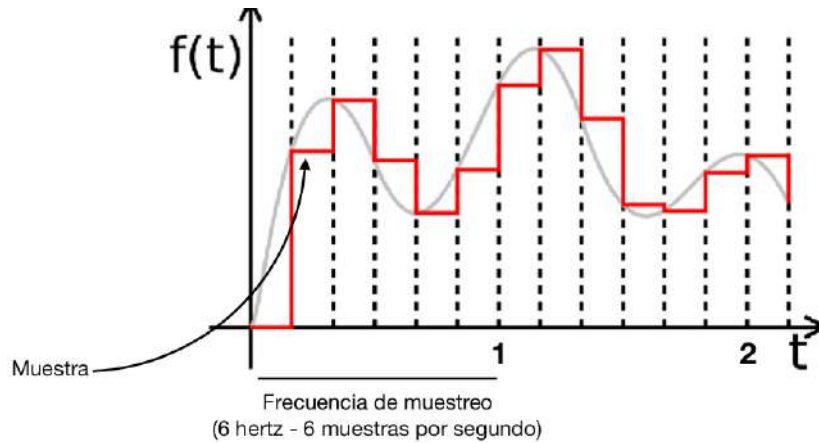
La parte central que estudiaremos acerca de la conversión de señales vive en nivel 0, y por lo tanto lo único que nos interesa es entender cómo un dispositivo puede tomar una señal analógica, y convertirla en una señal digital (y viceversa).

3.2. Información analógica

Una **señal analógica** es un tipo de señal generada por algún fenómeno, con una cierta **amplitud** y una **frecuencia**, que es representable por una función continua. Estas señales no pueden ser directamente procesadas por una computadora, sino que es necesario primero modificarlas.

El problema de que estas señales sean continuas es que no importa con cuánta frecuencia midamos estas señales analógicas, en el medio de dos mediciones tenemos infinitos valores, y además no tenemos manera de representar la totalidad de los valores posibles valores de estas señales.

Estos problemas se resuelven **digitalizando** las señales. Esta técnica consiste en limitar la cantidad de posibles valores que pueden tomar las señales, a partir de definir n umbrales (valor mínimo de una magnitud a partir del cual se produce un efecto determinado), suficientes para capturar la amplitud de las señales, y además fijar una frecuencia de muestreo, que resulte adecuada para no perder valores importantes.

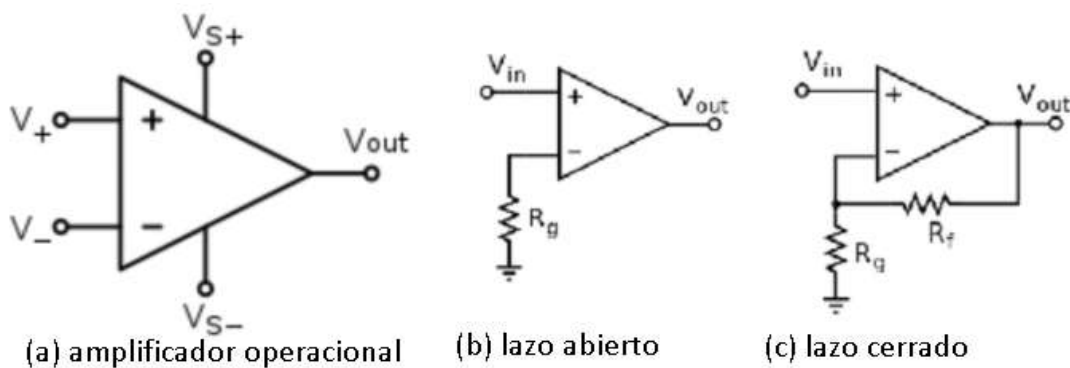


Un criterio (Nyquist criterion) para resolver este problema consiste en realizar un muestreo a una frecuencia mayor que el doble de la máxima frecuencia en la señal. Por ejemplo, la máxima frecuencia reconocible por el oído humano es 22 KHz, y por ello la frecuencia de muestreo sobre la que se distribuye el audio digital es 44 KHz.

3.3. Amplificador operacional

El elemento electrónico principal utilizado para convertir señales analógicas a señales digitales (y viceversa) es el **amplificador operacional**. Este tiene dos señales de entrada (V_- y V_+), y amplifica la diferencia de tensión entre esas dos señales ($V_{out} = AMP(V_+ - V_-)$).

Este componente electrónico nos permite diferenciar dos señales, por muy parecidas que sean. Estos dispositivos pueden conectarse de dos maneras: lazo abierto, lazo cerrado.



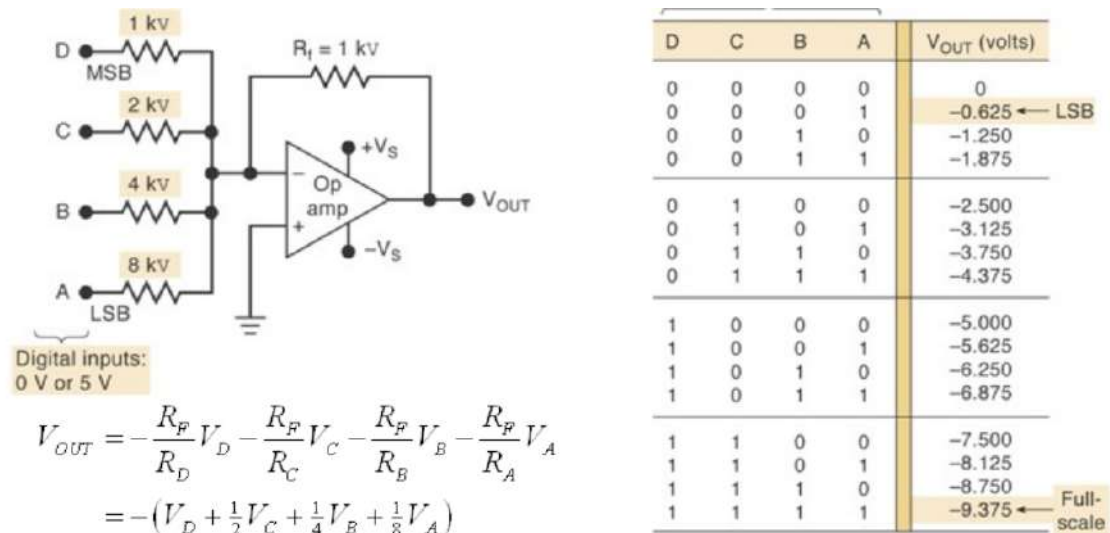
La configuración de **lazo abierto** es la que explicamos anteriormente, y se utiliza como comparador. En la configuración de **lazo cerrado** se retroalimenta la salida V_{out} a la entrada V_- , y de esta manera se reduce la diferencia entre V_- y V_+ , permitiendo distinguir entre valores intermedios.

Un comportamiento importante del amplificador operacional es que si V_+ es mayor o igual a V_- , devuelve en V_{out} 0 V. Por lo tanto, obtenemos $V_{out} = 0$ si $V_- \leq V_+$. Además, notemos que los amplificadores operacionales son circuitos caros de construir, por lo que trataremos minimizar su uso en la construcción de circuitería.

3.4. Conversiones Analógico-Digital y Digital-Analógico

3.4.1. Conversión Digital a Analógica

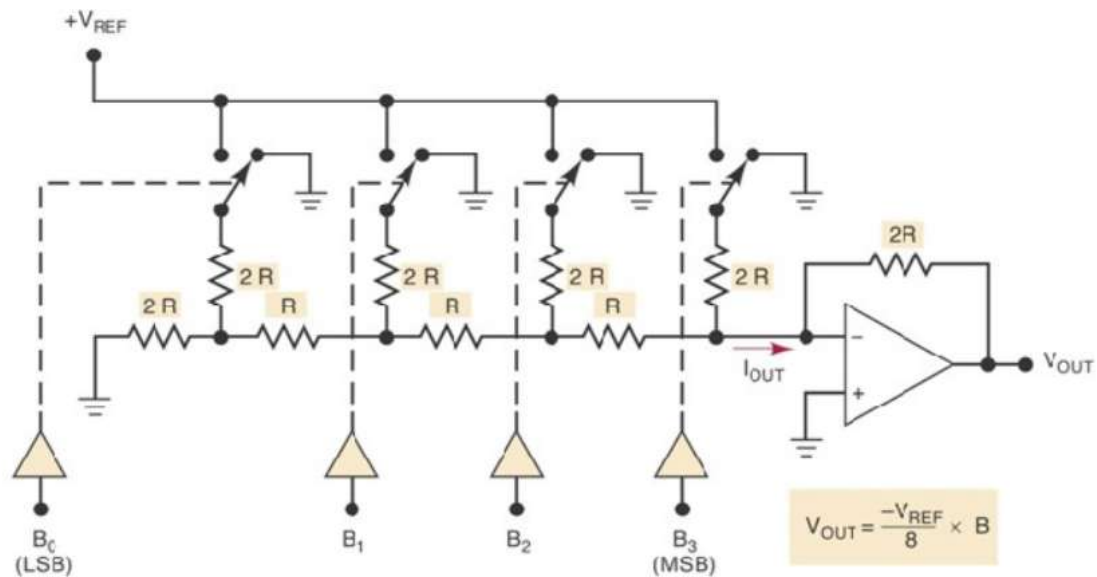
Primero veamos cómo pasamos de señales digitales a analógicas:



En este caso, representamos en el mundo digital a esta información mediante 4 bits (A, B, C, D , de menos significativo a más significativo). Notemos que las distintas combinaciones de estos bits terminan determinando los umbrales que discretizan a nuestra señal analógica. Por otro lado, notemos que V_+ está conectado a tierra, por lo que $V_{out} = 0$ si $A, B, C, D = 0$.

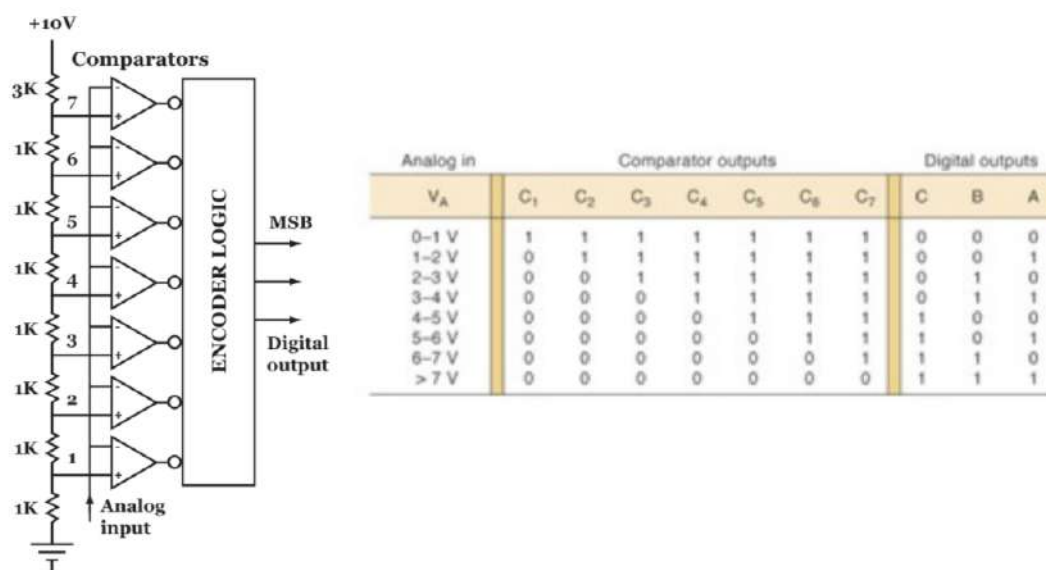
La idea es tomar estas 4 señales digitales (0V o 5V) independientes, y asignarles un peso relativo mediante el uso de resistencias (a menor peso, mayor la resistencia), para luego sumarlos, y obtener una señal analógica (en este caso, se obtiene una señal analógica entre 0V y 9.375V).

Existen otro tipos de circuitos que permiten independizar al voltaje de salida del voltaje de las entradas:



La idea es que nuestras señales digitales, en vez de funcionar como voltaje de entrada, terminan funcionando como señales de control, habilitando los distintos caminos por los que circula la señal de referencia $+V_{REF}$ (voltaje arbitrario). Notemos que no estamos aumentando la cantidad de umbrales, sino el voltaje representado por cada uno.

Lo que vimos hasta ahora sirve para convertir una señal que vive en la computadora (5V o 0V) a una señal analógica (con una cierta amplitud y frecuencia). Ahora veamos el proceso inverso, es decir convertir una señal analógica en una señal digital.



En el caso de que la entrada (*Analog Input*) sea mayor que los primeros n voltajes de la red resistiva

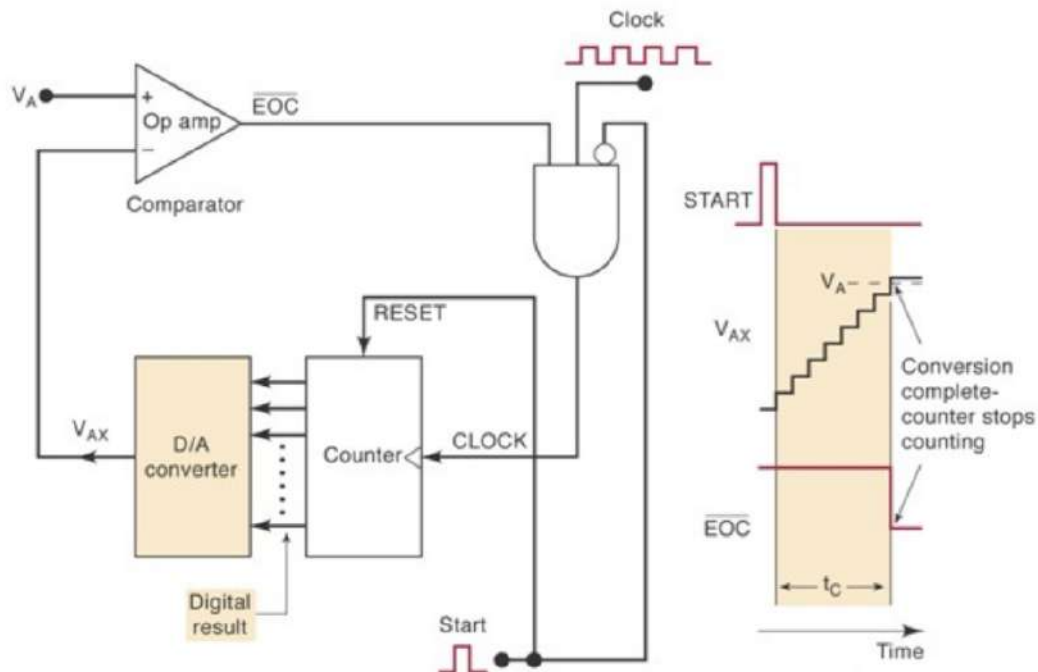
(y menor que todos los siguientes), entonces a partir del $n + 1$ el amplificador operacional devuelve 0. Luego, se niegan estas señales, para utilizarlas en el encoder, el cual devuelve la codificación del n -ésimo umbral en bits.

Este circuito es muy eficiente, al resolverse de manera electrónica, pero demasiado costoso debido a la cantidad de comparadores que se necesitan (uno por cada umbral posible). Si queremos tener 2^{16} valores representables, necesitamos 2^{16} amplificadores operacionales.

Para construirnos un circuito más eficiente, en lugar de tener un circuito que nos construye la señal digital a partir de comparaciones, podemos calcular esa señal a partir de sensar la señal analógica.

La idea es tener un circuito contador, al cual lo vamos a ir incrementando, y en cada iteración compararla con la señal analógica original (que queríamos convertir a digital). De esta manera, cuando nuestro contador alcance un voltaje mayor a la señal analógica, encontramos la representación digital de la misma (se levanta EOC, y nos guardamos el contenido del contador donde corresponda).

De esta manera, pasamos de necesitar 2^k amplificadores operacionales a solo 1.

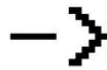


Capítulo 4

Memoria y Cache

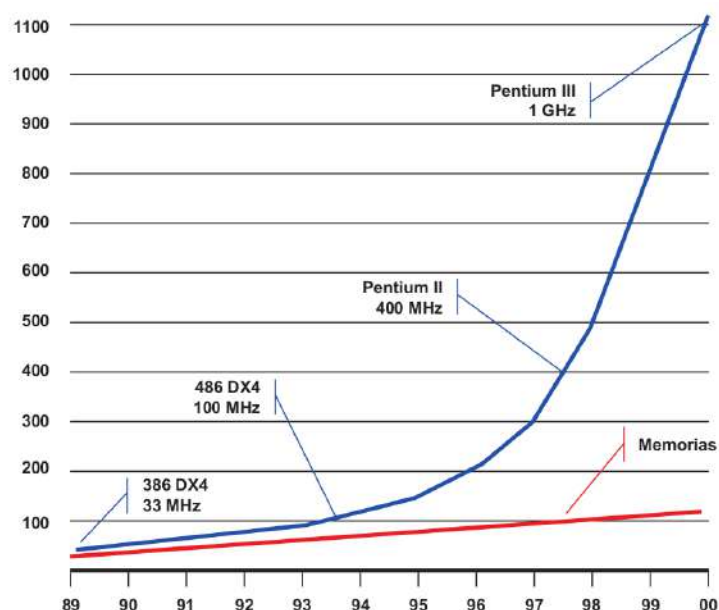
4.1. Introducción

Ya vimos cómo construir celdas de memoria en el capítulo 4, pero ahora vamos a estudiar los diversos tipos de memoria, y qué tecnología se utiliza para la construcción de las memorias, y cómo se organizan.



Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

Al momento de diseñar una memoria principal, tenemos que lidiar con el problema de la asimetría en la tasa de crecimiento de la velocidad de los procesadores con respecto a la tasa de crecimiento de la velocidad de las memorias.



Frente a este problema, se siguen introduciendo nuevas tecnologías en un intento de igualar las mejoras en el diseño de la CPU, ya que la velocidad de la memoria tiene que, de alguna manera, seguir el ritmo de la CPU, o la memoria se convierte en un cuello de botella, generando ciclos de espera para el procesador.

Luego, veamos qué tipos de memorias hay para poder elegir cuáles usaremos en la construcción de la memoria principal.

4.2. Tipos de Memorias

Aunque existe una gran cantidad de tecnologías de memoria, solo hay dos tipos básicos de memoria: **RAM** (random access memory) y **ROM** (read-only memory).

4.2.1. ROM

Este tipo de memorias se utilizan por la velocidad que tienen en las lecturas, y no importa qué tan lentas sean para escribir, ya que no se espera que este tipo de memorias sean escritas.

Las memorias ROM son no-volátiles, es decir que siempre conserva sus datos, incluso si apagamos el sistema. Además, son más baratas de construir, en grandes volúmenes, que las memorias RAM.

- **PROM:** Las PROM (Programmable ROM) contienen una serie de diminutos fusibles en su interior. Se puede fundir un fusible específico seleccionando su fila y columna, y luego aplicando un alto voltaje a un pin especial en el chip.
- **EPROM:** Las EPROM (Erasable PROM) se pueden borrar al exponerlas a una luz ultravioleta fuerte durante 15 minutos. Si se esperan muchos cambios durante el ciclo de diseño, las EPROM resultan más baratas que los PROM al ser reutilizables.
- **EEPROM:** Las EEPROM (Electrically EPROM) se pueden borrar aplicando pulsos eléctricos, en lugar de tener que utilizar una cámara especial para la exposición a los rayos ultravioleta. En el lado negativo, las EEPROM más grandes tienen típicamente solo 1/64 de capacidad con respecto a las EPROM, y además son la mitad de rápidas.

No podemos utilizar a las memorias ROM para construir la memoria principal, ya que el costo de programarlas es altísimo (comparado con las escrituras en las RAMs), incluso en el mejor caso de las EEPROM. Sin embargo, la mayoría de las computadoras contienen una pequeña cantidad de ROM (memoria de solo lectura) que almacena información crítica necesaria para operar el sistema, por ejemplo, el programa necesario para arrancar la computadora. Este tipo de memoria también se utiliza en sistemas integrados o cualquier sistema en el que no sea necesario cambiar la programación.

4.2.2. RAM

Memoria RAM es un nombre poco apropiado, ya que todas las memorias aceptan accesos aleatorios, pero la gran diferencia entre las RAM y las ROM es que estas permiten realizar escrituras en un tiempo similar a las lecturas, a costa de tener lecturas un poco más lentas y ser memorias volátiles, es decir que pierden la información que almacena una vez que se apaga.

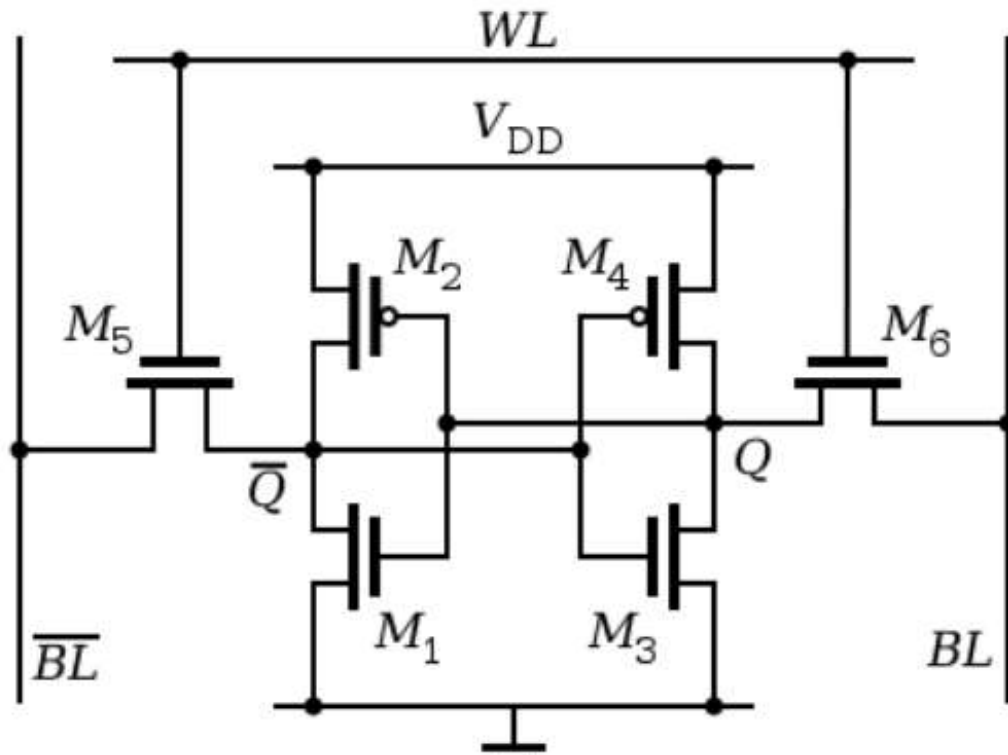
Principalmente hay dos tipos generales de chips que se utilizan para construir la mayor parte de la memoria RAM en las computadoras actuales: SRAM y DRAM (memoria de acceso aleatorio estática y dinámica). Veamos qué características tienen cada una para poder determinar cuál podría ser la función de cada una.

Es de interés estudiar las diferencias entre las SRAM y DRAM, evaluando las siguientes métricas:

- Capacidad de almacenamiento
- Tiempo de acceso (tanto para leer como para escribir)
- Velocidad de transferencia
- Consumo de energía
- Tamaño físico
- Costo total o por byte

SRAM

Vamos a empezar analizando la construcción de una memoria SRAM. Las memorias SRAM se parecen mucho a las celdas de memoria que estudiamos en Circuitos Secuenciales, y se caracterizan por ser construidas únicamente a partir de transistores, y tener una lectura / escritura muy eficiente.

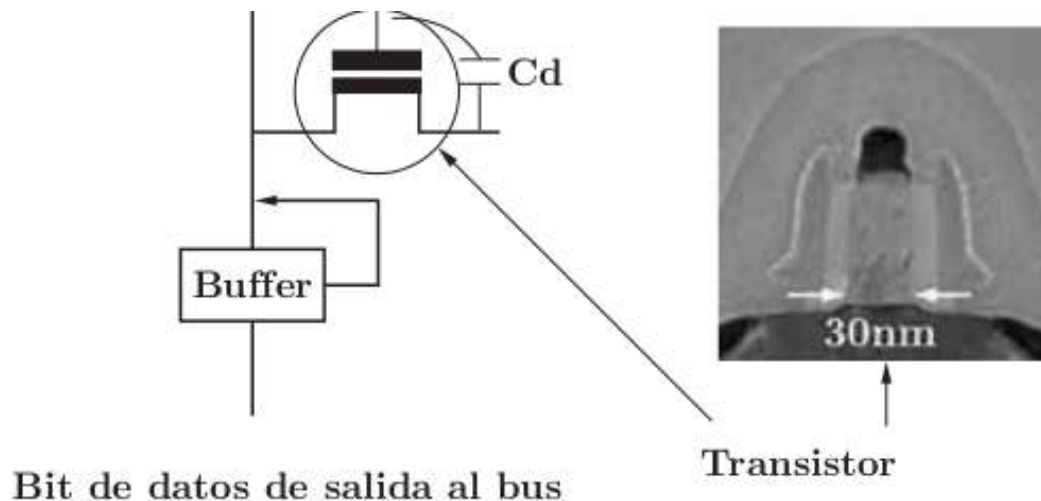


Este circuito es conocido como **biestable**, ya que garantiza que Q y \bar{Q} sean señales complementarias. Además, está construido a partir de 6 transistores, 3 de los cuales están siendo utilizados con energía máxima (los otros 3 son utilizados con energía residual), lo cual produce un gran consumo. Por otro lado, la lectura es directa (miro el valor de Q) y no destructiva (puedo leerlo cuantas veces quiera, sin afectar al estado del circuito), por lo que resultan muy rápidas.

El problema con este tipo de celdas es que se necesitan demasiados transistores, y por lo tanto este circuito genera mucho calor y tiene un alto consumo.

DRAM

Como el consumo de energía y el costo de fabricación aumentan considerablemente a medida que hacemos crecer la memoria, esto llevó a búsquedas de otras tecnologías para construir memorias. Frente a este problema, surgen las memorias DRAM, que son mucho más eficientes en cuanto a costo de almacenamiento (por bit).



En este diseño, un bit se almacena en un capacitor (una especie de batería recargable, en la que se puede almacenar una carga), de manera representar a los 0s como un capacitor descargado, y los 1s como un capacitor cargado.

El problema con este tipo de memorias es que la lectura es indirecta, ya que no estamos leyendo el bit (5V o 0V), sino que se lee una **carga residual** de un capacitor, la cual termina funcionando como señal de control, para activar una línea con la tensión correspondiente al bit guardado.

Por otro lado, este tipo de memorias requieren de una lógica interna mucho más compleja, a pesar de tener una construcción mucho más simple. Esto se debe a que tenemos que lidiar con dos problemas: la lectura es destructiva, y los capacitores pierden su carga a medida que pasa el tiempo.

Como la lectura es destructiva, se requiere de un mecanismo que refresque la carga del capacitor. Luego, cuando se lee una carga residual del capacitor, no solo se activa la línea que alimenta los 5V, sino que además hay que habilitar la recarga de ese capacitor, y por lo tanto las lecturas sucesivas de una celda se vuelven mucho más lentas.

Por otro lado, se necesita poder refrescar de forma periódica la carga de los capacitores, y así evitar perder la carga de los mismos. Esto requiere de un manejo de las cargas mucho más complejo, al tener que hacer lecturas y retroalimentaciones de forma periódica, lo cual también degrada la velocidad en las lecturas.

A pesar de estos problemas, logramos pasar de tener 6 transistores a un único transistor, el cual solo se activa cuando queremos hacer una lectura (y cuando tenemos que refrescar su información), y por lo tanto el consumo, el costo de fabricación, el tamaño de la celda, y la generación de calor son mucho menores.

Luego, las diferencias entre las SRAM y las DRAM son:

SRAM	DRAM
Alto consumo	Consumo mínimo
Capacidad de almacenamiento baja	Capacidad de almacenamiento alta
Costo por bit alto	Costo por bit bajo
Tiempo de acceso bajo	Tiempo de acceso alto

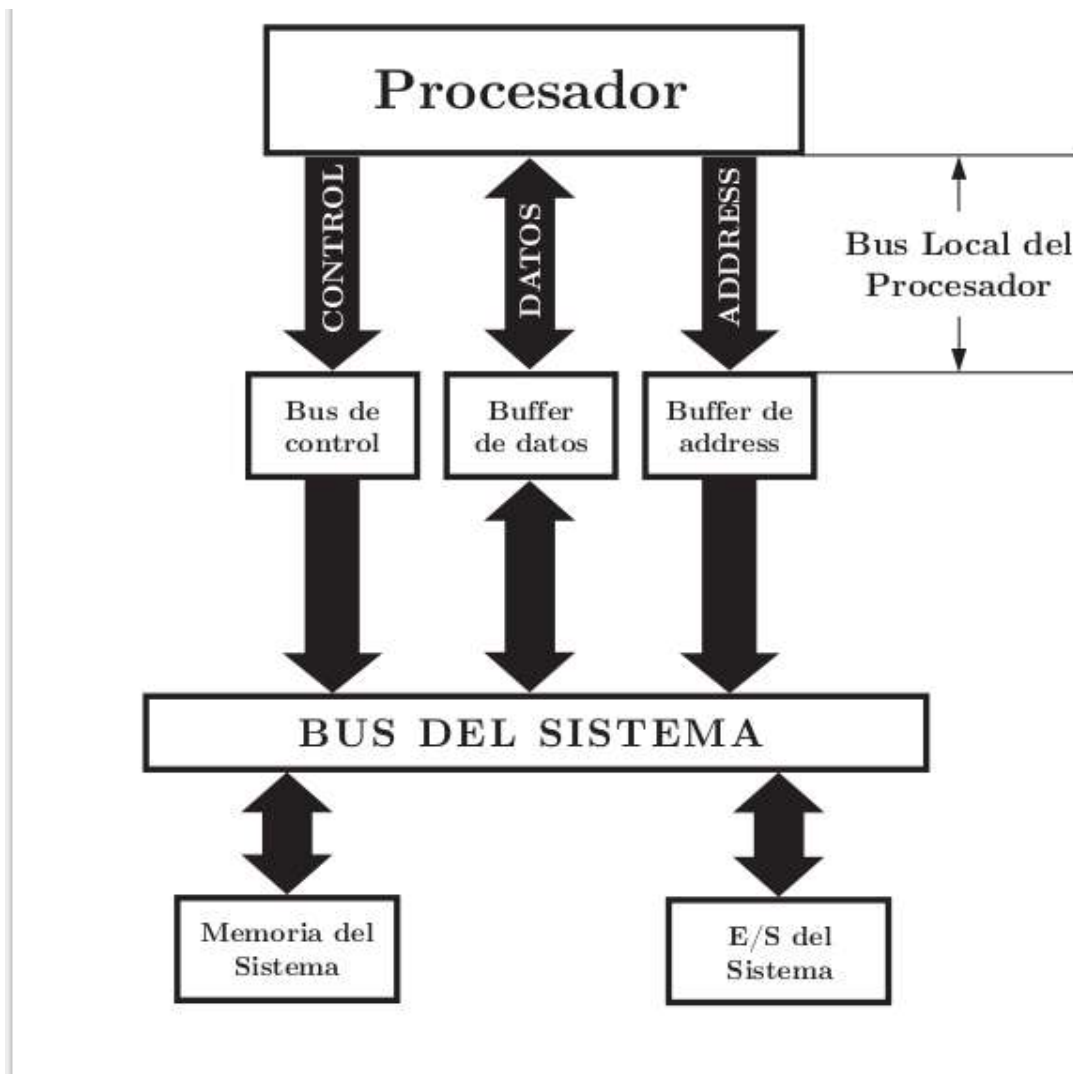
Cuadro 4.1: Comparación entre SRAM y DRAM

En resumen, si construimos el banco de memoria utilizando SRAM, el costo y el consumo (y sobre todo el calor) de la computadora se vuelven demasiado altos, y si lo construimos utilizando únicamente DRAM, desperdiciamos la velocidad del procesador, al volverse los accesos a memoria un cuello de botella.

Por lo tanto, ninguna de las dos propuestas solucionan el problema de la asimetría con respecto al crecimiento de la velocidad de los procesadores. Luego, tenemos que entender a un mayor nivel de profundidad cómo es que la memoria principal opera en relación al procesador, para poder llegar a una solución más ingeniosa.

4.3. Estructura de bus clásica

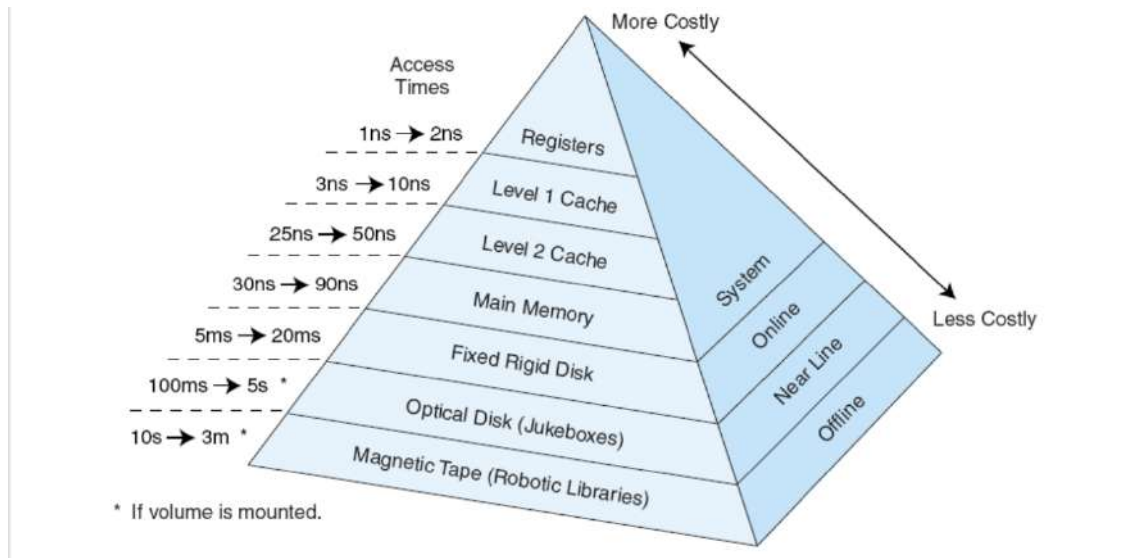
La estructura clásica en la que un procesador se conecta con una memoria o con un sistema de E/S consiste en conectar a la CPU con el resto de las componentes del sistema, a través de un bus que consta de tres tipos de señales: Control, Datos, y Direcciones.



Ya vimos que construir memorias de grandes volúmenes de información a partir de SRAM es inviable (principalmente por el calor que generan), y que utilizar memorias DRAM obligan al procesador estar esperando a que la memoria responda. Por lo tanto, necesitamos reconfigurar esta manera en la que se conecta la CPU con la memoria principal.

La forma de resolver este problema es construir una **jerarquía de memorias**. La idea es clasificar la memoria en función de su “distancia” del procesador, de manera que cuanto más cerca esté la memoria del procesador, más rápida debería ser.

Luego, a medida que la memoria se aleja del procesador principal, podemos permitirnos tiempos de acceso más largos, permitiendo el uso de tecnologías más lentas para estas memorias “lejanas” al procesador.



En general, en un sistema, vamos a encontrar todos estos tipos de memorias. La memoria más rápida la vamos a tener dentro del procesador, permitiendo que el procesador cuente con la información guardada en los **registros** prácticamente en tiempo real. Los registros son contruidos a partir de circuitos biestables (**SRAM**).

Luego están las **memorias cache**. La memoria cache de nivel 1 suele estar dentro del procesador, pero un poco más alejada que los registros, y aparece como un subsistema aparte dentro del mismo procesador, lo cual obliga a la CPU tener electrónica específica (el controlador de cache) que permita administrar la memoria cache.

Estas memorias cache también son contruidas a partir de circuitos biestables (**SRAM**), pero el costo de administración de las mismas hace que estas sean más lentas que los registros, sin embargo es lo suficientemente rápida para resolver el problema de tener costos excesivamente altos de accesos a la memoria principal del sistema.

La memoria cache de nivel 2 es auxiliar, y en general está situada sobre el motherboard, operando como puente entre el procesador y la memoria principal. Estas memorias cache también están contruidas con memorias **SRAM**, pero su administración es aún más compleja que la de nivel 1, y además se encuentra aún más lejos del procesador, por lo que tiene tiempos de acceso más lentos.

Luego tenemos la memoria principal del sistema, que se contruye con memorias **DRAM**, en la que vamos a almacenar un gran volumen de información. Por debajo están las memorias que se utilizan como memorias secundarias (discos rígidos, discos ópticos, cintas magnéticas), que utilizan otro tipo de tecnologías para almacenar información.

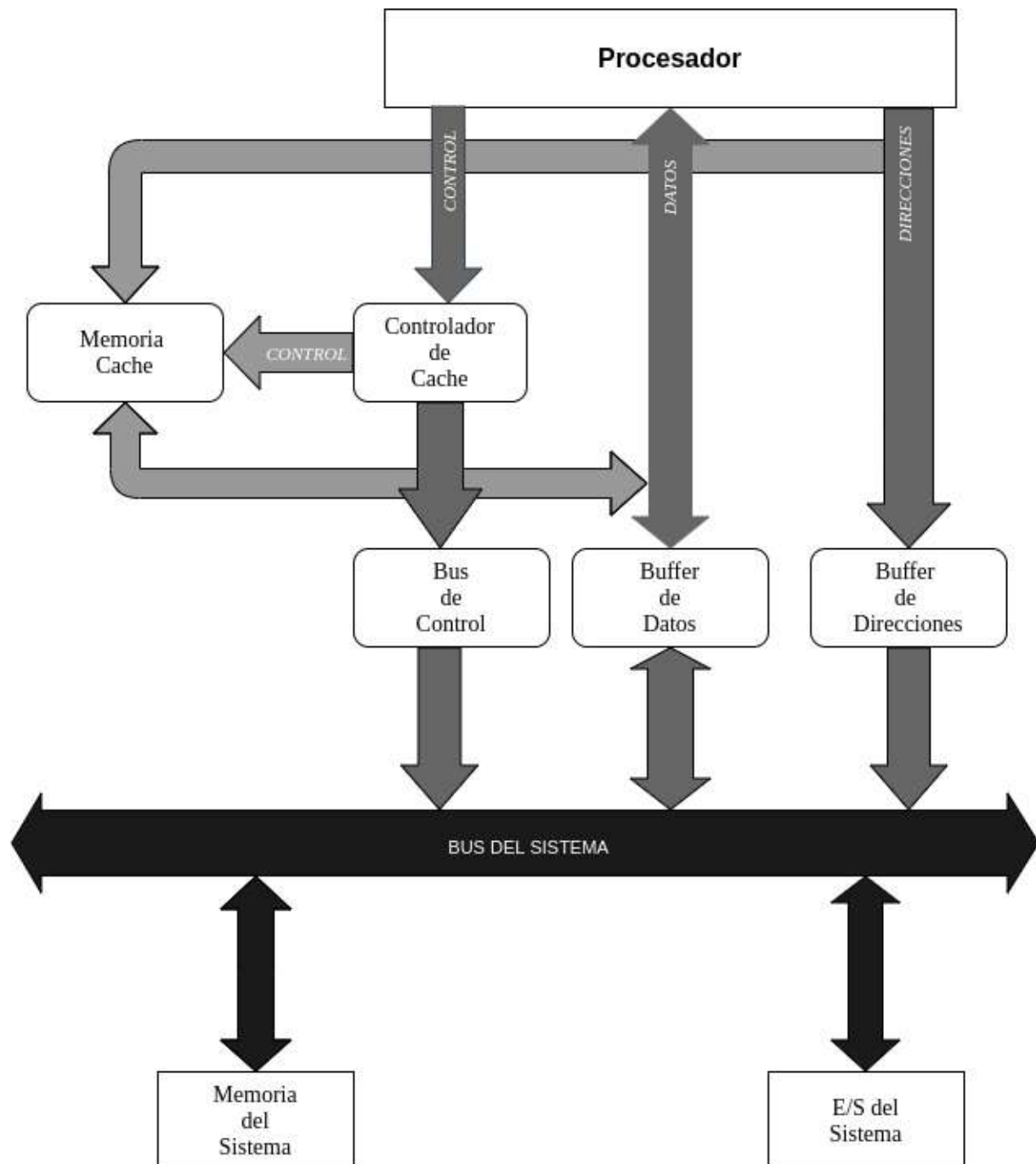
Al usar un esquema jerárquico de este tipo, se puede mejorar la velocidad de acceso efectiva de la memoria, utilizando sólo una pequeña cantidad de chips rápidos (y costosos). Esto nos permite crear una computadora con un rendimiento aceptable a un costo razonable.

4.3.1. Estructura de bus con cache

Como las memorias cache son mucho más pequeñas que la memoria principal, tenemos que poder guardar en ellas únicamente una copia de los datos e instrucciones más importantes (que serán accedidos muchas veces), y así poder asegurar una alta performance.

Además, se necesita de algún mecanismo que permita saber qué es lo que hay almacenado en las cache, para que se pueda devolver el dato de la cache correspondiente a la dirección de memoria a la que intenta acceder el procesador.

Por otro lado, se necesita de hardware adicional que asegure la consistencia de los bancos de cache, con respecto a los datos almacenados en la memoria principal.



La idea va a ser colocar una memoria cache entre el procesador y la memoria principal del sistema, engañando al procesador, para que cuando éste pida un dato en memoria, el controlador de la cache busca en la cache si tiene el dato, y si lo tiene (**hit**) se lo devuelve a la CPU, y nos ahorramos tener que acceder a la memoria principal.

Si no tiene el dato (**miss**), el controlador de la cache busca el dato en la memoria principal, para luego refrescar la cache, guardar el conjunto correspondiente al dato, ocupando el lugar de otro conjunto (menos relevante, bajo algún criterio), y enviarlo a la CPU. Luego, podemos definir la eficacia de una memoria cache a través del **hit rate** = $hit / \#accesos$.

Para optimizar el uso de una memoria cache, es decir optimizar el *hit rate*, existen dos principios: el de **vecindad espacial**, y el de **vecindad temporal**. El **principio de vecindad espacial** consiste en suponer que si un dato es utilizado, es probable que los datos cercanos en la memoria sean utilizados pronto (por ejemplo, las distintas posiciones de un arreglo o las variables locales de un programa).

El **principio de vecindad temporal** consiste en que si un dato es utilizado, es probable que sea accedido nuevamente en un futuro cercano (por ejemplo, variables utilizadas como contadores de un ciclo).

Luego, bajo estos dos principios, nos interesa almacenar en la cache no solo las variables que hayamos usado más recientemente, sino que también aquellas que se encuentren cercanas a estas.

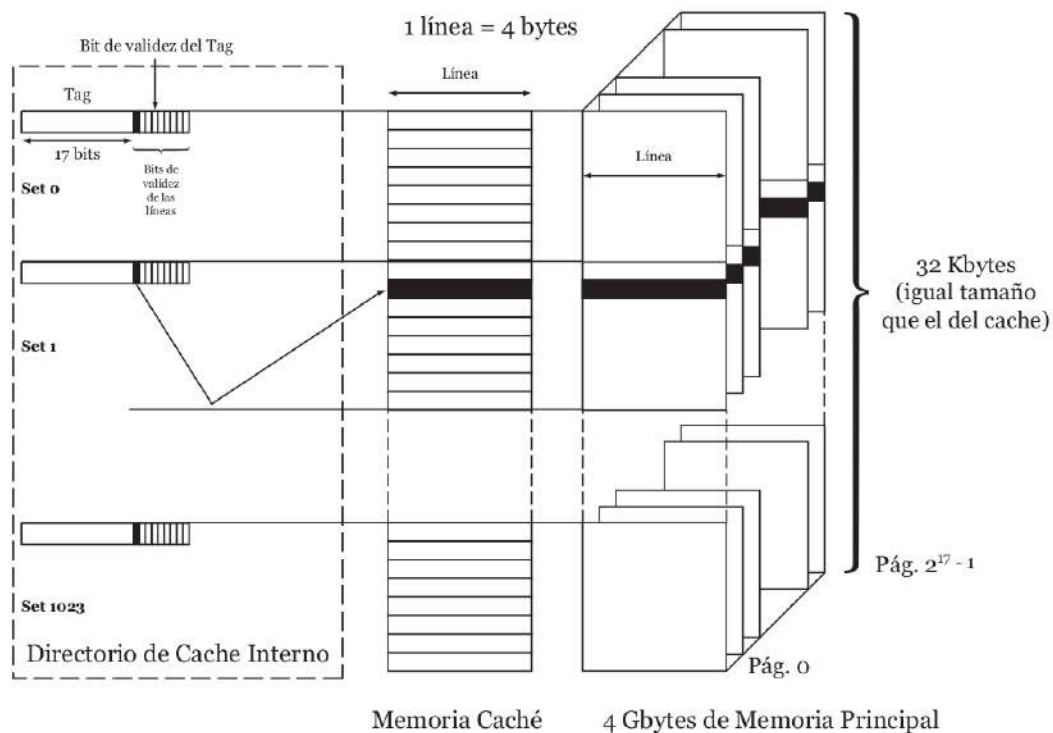
4.4. Organización de la Cache

Para que la cache sea funcional, debe almacenar datos útiles. Sin embargo, estos datos se vuelven inútiles si la CPU no puede encontrarlos. Al acceder a datos o instrucciones, la CPU primero envía por el bus de direcciones la dirección de la memoria principal a la que quiere acceder.

El problema está en el hecho de que dentro de la cache, la dirección de los datos no es la misma que la dirección de la memoria principal. Luego, necesitamos un mecanismo que nos permita “convertir” las direcciones con las que se maneja la CPU a la ubicación en la memoria cache.

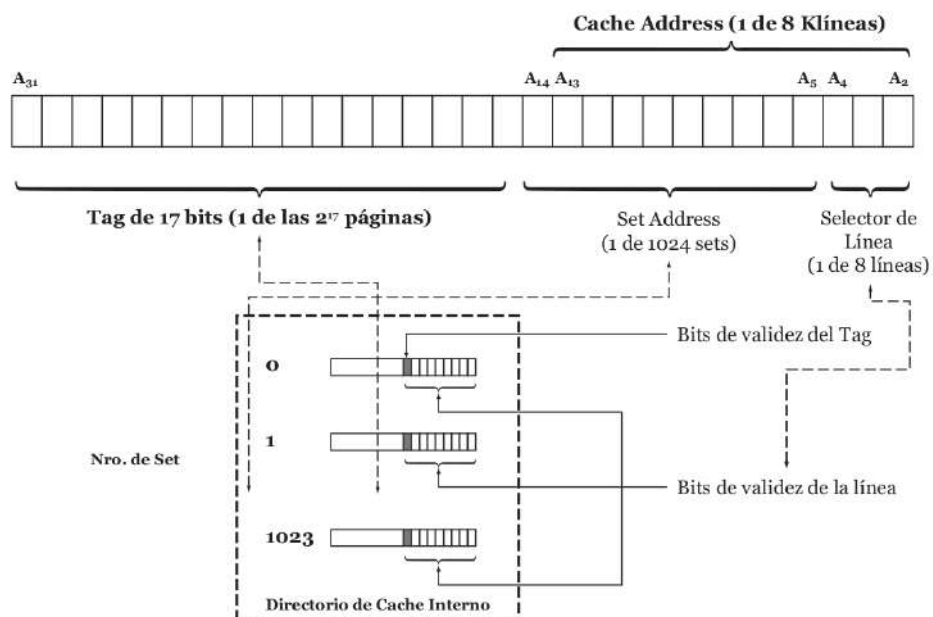
Lo primero que tenemos que hacer es determinar el tamaño de la memoria cache y el de la memoria principal. La idea es dividir a la memoria principal en fragmentos (del mismo tamaño que la cache) que llamaremos **páginas**, y luego subdividir cada página en **conjuntos**, formados por un conjunto de **líneas** que son un múltiplo del tamaño de la palabra de datos de memoria, indexadas mediante un **índice** que permite ubicar una palabra dentro de cada línea.

Al momento de traernos información de la memoria principal a la memoria cache, en vez de traernos únicamente una palabra, nos traeremos un conjunto completo, y de esta manera resolvemos el principio de vecindad espacial.



En este caso, la memoria principal es de 4 GB, la memoria cache es de 32 KB, los conjuntos son de 8 líneas, las líneas son de 4 bytes, por lo que tenemos 1024 conjuntos por cada página de 32 KB ($2^5 * 2^{10} / 2^2 = 2^{13} = 8 * 2^{10}$), y tenemos 2^{17} páginas en toda la memoria ($2^2 * 2^{30} / 2^5 * 2^{10} = 2^{17}$).

Luego, podemos obtener la siguiente información únicamente mirando la dirección (30 bits) de la memoria principal en la que se encuentra nuestro dato:



Los primeros 17 bits nos indica de qué página es el dato. Los 10 bits siguientes nos indican el conjunto dentro de la página. Los siguientes 3 bits indican la línea en la que se encuentra el dato, y

suponemos que las palabras son de 32 bits, por lo que no es necesario utilizar un índice (las líneas son de 1 palabra). (*) Nota: a partir de este punto vamos a suponer que los conjuntos son de 1 línea en vez de 8. También hay un tema con los nombres de las cosas (en la Teórica se habla de páginas, conjuntos, líneas, y palabras indexadas con un índice; en la Práctica se habla de bloques (líneas de la memoria), líneas (conjuntos de 1 línea), y palabras indexadas con un índice; y en el Null se habla de bloques (conjuntos), y palabras indexadas con un índice).

4.5. Esquemas de Mapeo

Para poder saber qué datos tenemos guardados en la cache, agregamos un **directorio de cache** (construido a partir de memorias SRAM), el cual guarda tiras de bits que caracterizan las líneas que tenemos almacenadas en la cache, a las que llamaremos *tag*. Estos tags tienen un formato específico, el cual permite identificar a qué fragmento de la memoria principal corresponde cada línea.

Dónde y cómo guardamos los datos, y qué formato tiene el tag va a depender del esquema de mapeo de la cache. Vamos a estudiar tres alternativas: **Mapeo Directo**, **Mapeo asociativo de N vías**, **Mapeo completamente asociativo**.

4.5.1. Mapeo Directo o de Correspondencia Directa

El esquema de Mapeo Directo consiste en dividir la memoria principal en páginas del tamaño de la cache. Luego, cada línea de cada la página tiene un lugar específico en la cache, y por lo tanto, solo tenemos que almacenar como tag, en el directorio de la cache, la página a la que pertenece el dato.

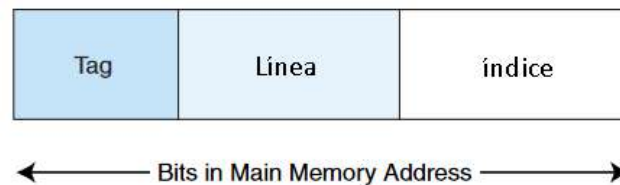


FIGURE 6.4 The Format of a Main Memory Address Using Direct Mapping

Para recuperar el dato, únicamente se necesita buscar la línea en la que debería estar guardado, y revisar si tiene el tag correspondiente, y en caso de que coincidan, indexarlo y obtener el dato.

Este esquema es el más barato, ya que no necesita de ninguna búsqueda dentro de la cache (ni de un HW específico para realizar todas las búsquedas a la vez). Cada bloque de memoria principal tiene una única ubicación a la que se asigna en cache, y por lo tanto, cuando una dirección de memoria principal se convierte a una dirección de cache, el controlador de la cache sabe exactamente dónde tiene que buscar.

El problema con este esquema es que si llega un nuevo dato, y la posición correspondiente en la cache ya estaba ocupada, se elimina la línea que estaba en la cache, y por lo tanto no podemos almacenar una misma línea de distintas páginas. Por ejemplo, no podemos tener la línea 5 de la página 3, y al mismo tiempo la línea 5 de la página 10, a pesar de tener espacio libre en la cache. Esto puede llevar a que con solo dos accesos a datos en la misma línea, pero de bloques distintos, constantemente se generen miss, al estar ambas líneas en conflicto por el mismo espacio en la cache.

4.5.2. Mapeo Completamente Asociativo

En el esquema de mapeo completamente asociativo, en lugar de especificar una ubicación única para cada línea (módulo página) de la memoria principal, se permite que cada línea de memoria principal se coloque en cualquier lugar de la cache. Luego, para poder identificar mediante el tag cada dato, necesitamos saber a qué página pertenece, y también a qué línea dentro de la página.

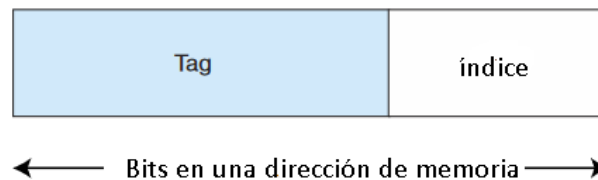


FIGURE 6.8 The Main Memory Address Format for Associative Mapping

Con esto logramos evitar el escenario anterior, ya que todas las líneas pueden ser almacenadas en cualquier parte de la cache. Sin embargo, al momento de hacer una lectura, hay que comparar con todos los tags del directorio, ya que la línea puede estar en cualquier parte de la cache, y por lo tanto es más lento (o requiere de un HW específico para poder realizar todas las búsquedas al mismo tiempo, resultando en un esquema más costoso).

4.5.3. Mapeo Asociativo por Conjuntos de N vías

Este esquema es similar al mapeo directo, ya que usamos la dirección para mapear el bloque a una determinada ubicación de la cache. Sin embargo, la diferencia importante es que en lugar de mapear a una sola línea de cache, cada dirección se mapea a un conjunto de N vías de cache.

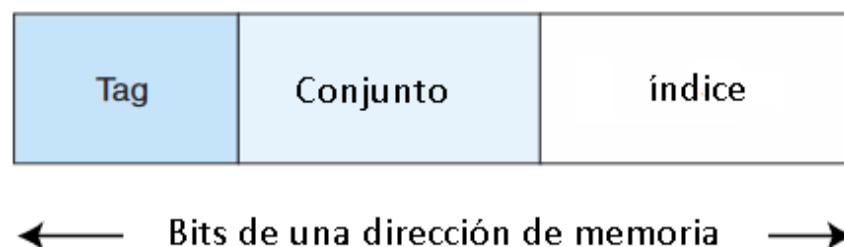


FIGURE 6.10 Format for Set Associative Mapping

De esta manera, cuando hacemos una búsqueda de un dato en la cache, no tenemos que revisar toda la cache, sino que solo tenemos que revisar las N vías correspondientes a al dato, resolviendo el problema que teníamos en Mapeo Directo, sin ser tan lento o costoso como el Mapeo Completamente Asociativo. Sin embargo, tiene un mayor costo (por el HW específico para realizar búsquedas en simultáneo) y el tamaño de la vía es N veces más chica que la línea en Mapeo Directo (para un mismo tamaño de cache).

4.6. Políticas de reemplazo de contenido

Cuando estamos trabajando bajo el esquema de Mapeo Directo, si hay un conflicto por un espacio en la cache, solo hay que reemplazar la línea vieja por la nueva (vecindad temporal). No hay necesidad de ninguna política de reemplazo, ya que la ubicación del nuevo bloque es predeterminada.

Sin embargo, cuando trabajamos con un esquema asociativo, ya sea completamente asociativo o asociativo por conjuntos de N vías, es necesario tener un algoritmo de reemplazo para determinar qué línea de la cache hay que reemplazar por la nueva.

Cuando trabajamos con un esquema completamente asociativo, se podría reemplazar cualquiera de las K líneas de la cache, mientras que con esquema de N vías, tenemos solo N opciones.

Least Recently Used (LRU)

Podríamos suponer que es más probable que se vuelva a necesitar un dato que fue accedido recientemente con respecto a uno que no. Luego, la idea de LRU es realizar un seguimiento de la última vez que se accedió a cada línea de la cache, guardando el *timestamp* en el directorio, y remover aquella línea (o vía) que se haya utilizado menos recientemente.

Desafortunadamente, LRU requiere que el sistema mantenga un historial de accesos para cada línea (o vía) de la cache, lo que requiere un espacio significativo y ralentiza el funcionamiento de la cache.

Least Frequently Used (LFU)

La idea del esquema LFU consiste en que el sistema lleve un registro del número de veces que se hace referencia a una línea en la memoria. Cuando la cache está llena, y requiere más espacio, el sistema removerá la línea (o vía) con la frecuencia de referencia más baja.

Este esquema se implementa asignando un contador a cada línea (o vía) que se carga en la cache. Cada vez que se hace una referencia a esa línea, el contador aumenta en uno. Cuando la cache está llena, y tiene una nueva línea que debe ser guardada, el sistema buscará la línea (o vía) con el contador más bajo, y lo reemplazará por la línea entrante, reiniciando el contador.

First In - First Out (FIFO)

El esquema de FIFO es otro enfoque popular. Con este algoritmo la línea (o vía) que ha estado en la cache por más tiempo sería la próxima en ser reemplazada.

Random

El problema con LRU y FIFO es que hay situaciones en las que pueden causar *trashing*, es decir que tiran constantemente una línea, luego la traen de vuelta, repetidamente. Algunas personas argumentan que el reemplazo *random*, aunque a veces arroja datos que se necesitarán pronto, nunca genera *trashing*. Desafortunadamente es difícil tener un reemplazo verdaderamente random, y además puede disminuir el rendimiento promedio.

4.7. Políticas de Escritura

Una variable que está en la cache también está alojada en alguna dirección de la memoria principal. Ambos valores deben ser consistentes. Cuando el procesador modifica una variable hay varios modos de actuar:

Write through

La escritura se realiza tanto en la cache como en la memoria principal. Esto es más lento que la *write-back*, pero asegura que la cache sea consistente con la memoria principal del sistema. Si bien la escritura

requiere un acceso a la memoria principal, lo que ralentiza el sistema, en aplicaciones reales, la mayoría de los accesos son de lectura, por lo que esta ralentización es insignificante.

Write through buffered

El procesador actualiza la cache, y el controlador de la cache eventualmente actualiza el dato en la memoria (típicamente cuando ocurre un miss), mientras el procesador continúa ejecutando usando los datos actualizados de la cache.

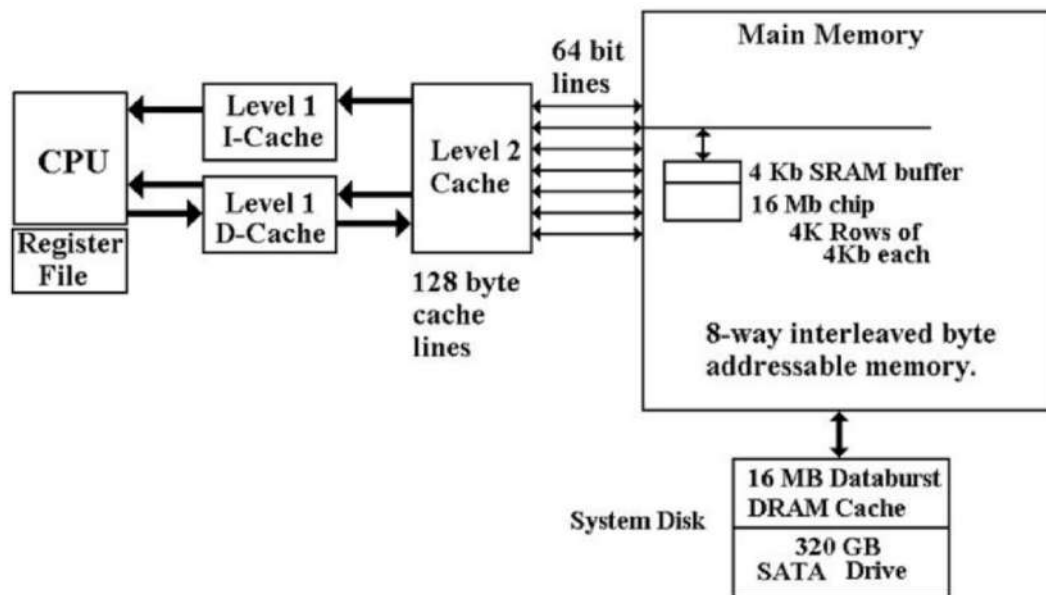
Copy back

Se marcan por HW las líneas de la memoria cache cuando el procesador escribe en ellas (parecido al bit Dirty de Intel). Luego, al momento de eliminar esa línea de la cache, el controlador de la cache deberá actualizar el valor en memoria. El problema de este esquema es que si un proceso falla antes de que se realice la escritura en la memoria principal, es posible que se pierdan los datos almacenados en la cache.

4.8. Cache Multinivel

Por último, veamos cómo está construida una pirámide de memoria. Sabíamos que teníamos dos niveles de memorias cache independientes (con sus propios controladores), una más pequeña dentro del procesador (L1 cache), y la otra más grande en el motherboard (L2 cache).

Las L1 cache suelen estar divididas en dos: una para datos y una para instrucciones. De esta manera, aprovechamos la vecindad espacial y la vecindad temporal para los dos tipos de accesos que realizamos desde el procesador (o bien estamos en el ciclo de fetch buscando la próxima instrucción, o bien estamos accediendo a un dato).



Capítulo 5

Buses

5.1. Introducción

Ya vimos cómo funcionan las componentes, el procesador, el sistema de E/S, y la memoria. Nos falta ver cómo es que estas componentes se conectan entre sí, e incluso con otros dispositivos para poder realizar intercambios de información, a nivel de HW.



Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

La implementación concreta de una computadora se basa en el hecho de que las componentes están conectadas a un único canal, por el que fluyen los datos. El hecho de tener un único canal, compartido por todas las componentes, se conoce como el **cuello de botella de Von Neumann**, ya que solo una componente lo puede usar en un determinado momento, y el resto tiene que esperar a que el bus se libere.

Si cada vez que tenemos que mandar un dato, tenemos que tomar el único canal que existe, entonces ese recurso va a condicionar en gran parte el rendimiento del sistema, y por lo tanto la manera en la que se administra este canal va a ser de gran importancia.

Si bien en el modelo de Von Neumann - Turing se plantea la existencia de un único canal que conecta la totalidad de las componentes, si queremos que nuestro sistema sea escalable (que podamos ir agregando nuevas componentes), este bus único crecería de manera cuadrática, por lo que resulta inviable su construcción.

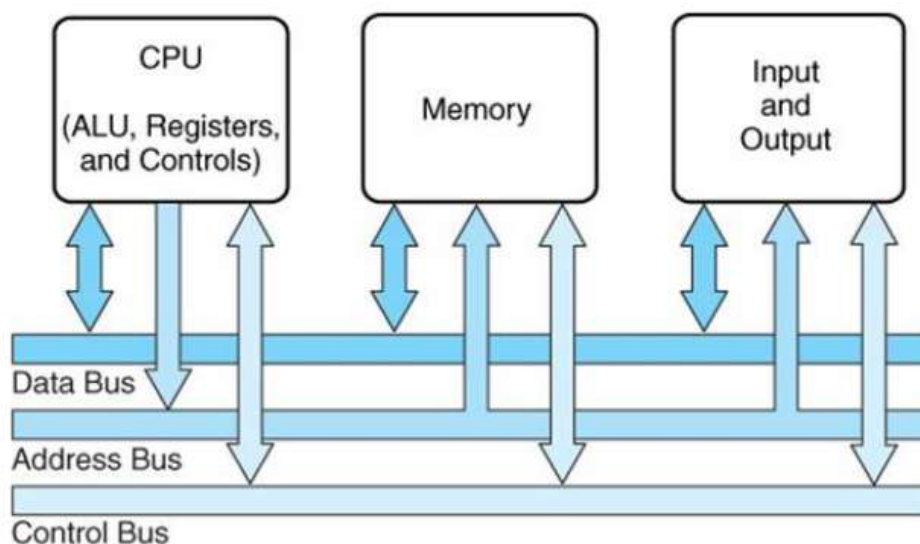
Además, queremos que estas conexiones estén estandarizadas, para facilitar la comunicación entre las distintas componentes, y facilitar la lógica necesaria para agregar nuevas componentes.

5.2. Buses

Un bus es un camino (cable) por el cual pasan señales eléctricas, que interconectar a múltiples dispositivos, por lo que son un recurso compartido.

En general los buses tienen diferentes tipos de señales. Podemos pensar que tenemos un bus con señales de propósito diferenciado o que tenemos diferentes buses que cumplen un rol distinto en el momento de la comunicación. Típicamente, hay tres roles de importancia: las señales de control, de datos, y de direcciones.

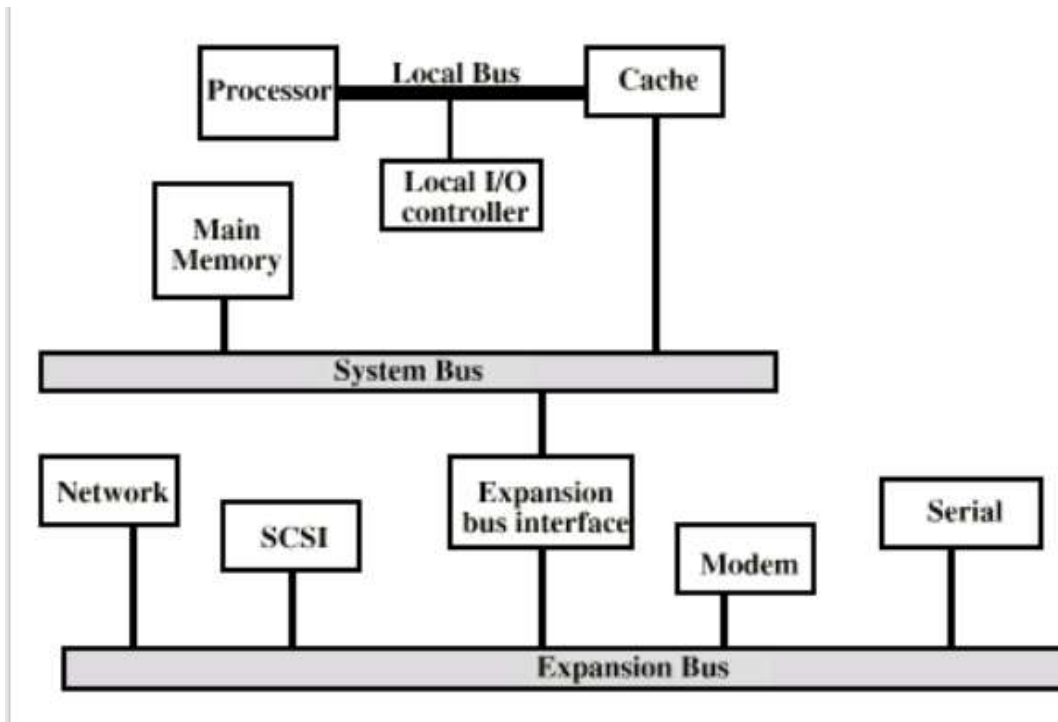
Las **señales** de dirección permiten identificar elementos puntuales en un subsistema (posiciones de memoria, o posiciones de E/S, etc). El **bus de datos** permite la transferencia de datos de un lugar a otro (el procesador recibe datos de la memoria, o la memoria recibe datos de E/S, etc). Las **señales de control** nos permiten administrar la información, para que las componentes que se comunican puedan tomar acciones sobre por dónde pasa la información (se inicia un ciclo de lectura, uno de escritura, se envían datos de un dispositivo a otro, etc).



Además, los buses se suelen estructurar de forma jerárquica. Recordemos que tenemos un sistema conformado a partir de varios subsistemas, que pretendemos que tengan un rol distinguido, y por lo tanto tenemos que tener en cuenta que el tiempo con los que se maneja la CPU son distintos a los que se maneja la memoria principal, que son distintos a los tiempos con los que se manejan los dispositivos de E/S.

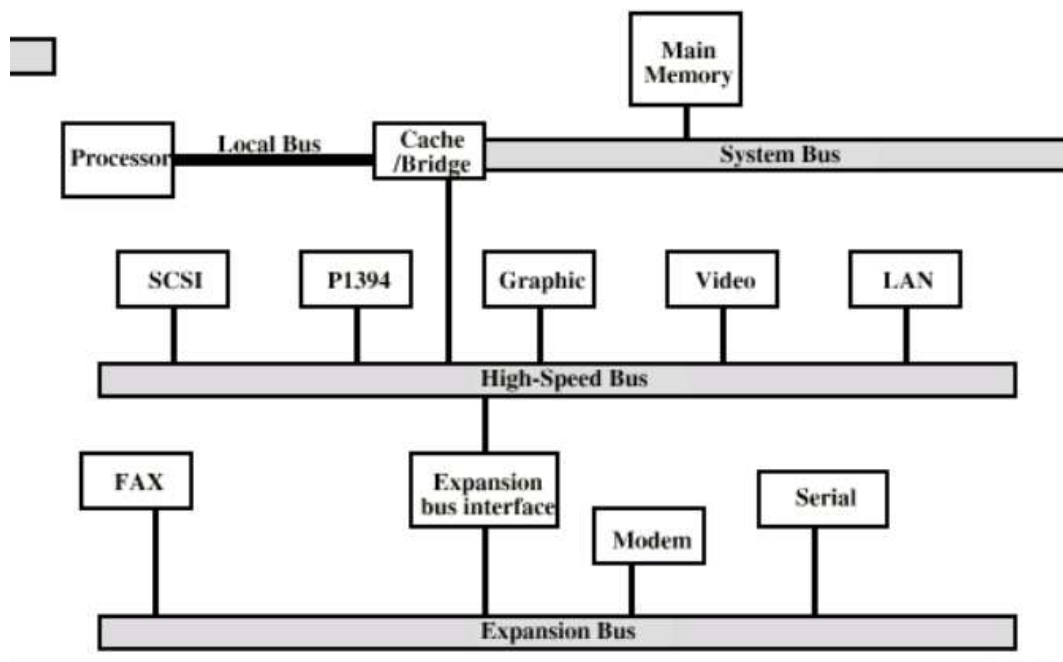
A medida que nos alejamos del procesador, los buses adquieren propósitos más específicos y pierden accesibilidad al procesador. Luego, podemos estructurar las conexiones de los buses de forma jerárquica, por ejemplo, conectando al procesador con la cache con un bus de alta velocidad, y los distintos dispositivos a un bus más lento.

No hay una única manera de construir esta jerarquía de buses, pero sí hay mecanismos por los cuales podemos pasar de un bus de alta velocidad a uno de menos velocidad. En este ejemplo, tenemos una jerarquía de buses de 3 niveles.



El **Bus Local** (más rápido) conecta a la CPU con la memoria cache, el **Bus del Sistema** (intermedio) conecta a la cache con la memoria principal, y el **Bus de Expansión** conecta los distintos dispositivos de E/S a la interfaz del bus de expansión. La conexión entre buses de distintos niveles se conoce como **bridge**, y funciona como interfaz entre un bus y otro.

Otra estructura de bus podría ser la siguiente:



En este caso tenemos un bus de alta velocidad para dispositivos privilegiados que tienen un alto rendimiento para la transferencia de datos (discos de alta velocidad, placas aceleradoras gráficas, placas de red).

5.3. Diseño de un bus

Si bien los diseños de la CPU pueden usar cualquier tipo de bus que se desee dentro del chip, para que las placas diseñadas por terceros puedan conectarse al bus del sistema, debe haber reglas bien definidas sobre cómo funciona el bus, al que todos los dispositivos conectados a él deben obedecer.

Estas reglas se denominan **protocolo** de bus. Además, debe haber especificaciones mecánicas y eléctricas, para que las placas de terceros quepan en la jaula de tarjetas y tengan conectores que coincidan mecánicamente, en términos de voltajes, y de temporización con los de la placa base.

Los principales problemas de diseño de los buses son el ancho del bus, la sincronización del bus, el arbitraje del bus y el tipo de línea. Cada uno de estos problemas tiene un impacto sustancial en la velocidad y el ancho de banda del bus. Ahora examinaremos cada uno de estos en las próximas cuatro secciones.

5.3.1. Ancho del bus

Uno de los elementos significativos a la hora de discutir buses es el **ancho del bus**, es decir la cantidad de líneas físicas con las que cuenta un bus para desarrollar su rol en el sistema.

Notemos que el ancho del **bus de direcciones** determina el espacio máximo direccionable de memoria, mientras que el ancho del **bus de datos** determina el tamaño de la palabra del sistema, y por lo tanto, la cantidad de accesos a memoria necesarios para transferir la misma cantidad de información.

El problema es que los buses anchos necesitan más cables que los buses estrechos. También ocupan más espacio físico (por ejemplo, en la placa base) y necesitan conectores más grandes. Todos estos factores encarecen el bus. Por lo tanto, existe un *trade-off* entre el ancho del bus y el costo del sistema.

5.3.2. Tipo de línea

Otro elemento importante es la naturaleza de las líneas de comunicación, que funcionarán como conexiones entre las distintas componentes y dispositivos. Las líneas de un bus se dicen que son **dedicadas** cuando es posible asignarles un rol en el sistema de forma que este sea siempre el mismo.

Existen dos tipos de dedicaciones: la **dedicación física** y la **dedicación funcional**. La dedicación física se enuncia cuando una línea particular tiene un rol específico en conectar un conjunto determinado de componentes. La dedicación funcional refiere a la tarea que desempeña la línea, y no tanto a una conexión física concreta.

Dividir las líneas del sistema según la dedicación implica menor cantidad de disputas, aumentando el rendimiento del sistema, ahorrando ciclos de reloj, pero al mismo tiempo incrementa el tamaño y costo del sistema. Por ejemplo, veamos cómo se realizaría una escritura con buses dedicados:

1. La componente A (que escribe) coloca la dirección en el bus de direcciones y los datos en el bus de datos en un mismo ciclo de reloj del bus
2. La componente A notifica de este hecho a la componente B

Para evitar el problema de los buses muy anchos, a veces los diseñadores optan por un **bus multiplexado**. En este diseño, se comparten las mismas líneas para realizar distintas transferencias. Por ejemplo, podemos utilizar las mismas líneas físicas para transferir direcciones y datos. Para una escritura en la memoria, por ejemplo,

Por otro lado, aunque no de manera ortogonal, podemos **multiplexar** las líneas, es decir reutilizar las líneas para que estas cumplan más de un rol de un sistema, dependiendo del momento particular de la comunicación, distinguidas a partir de un **protocolo de comunicación**, permitiendo que las distintas

componentes involucradas en la comunicación puedan reconocer en qué etapa se encuentra la comunicación. La ventaja de multiplexar las líneas es que se reduce el ancho del bus (y el costo), pero esto genera transferencias más lentas.

Un uso típico de multiplexación es utilizar bus de datos y de direcciones sobre las mismas líneas físicas. Por ejemplo, veamos cómo se realizaría una escritura. Para que se pueda realizar la escritura, las líneas de dirección deben configurarse y propagarse a la memoria antes de que los datos se puedan colocar en el bus, lo cual no era necesario con líneas dedicadas:

1. La componente A (que escribe) coloca la dirección en el bus de direcciones y lo notifica a la componente B en un ciclo de reloj del bus
2. La componente B notifica que la dirección ya fue leída
3. A coloca los datos en el bus de datos en otro ciclo de reloj del bus
4. A notifica de este hecho a la componente B.

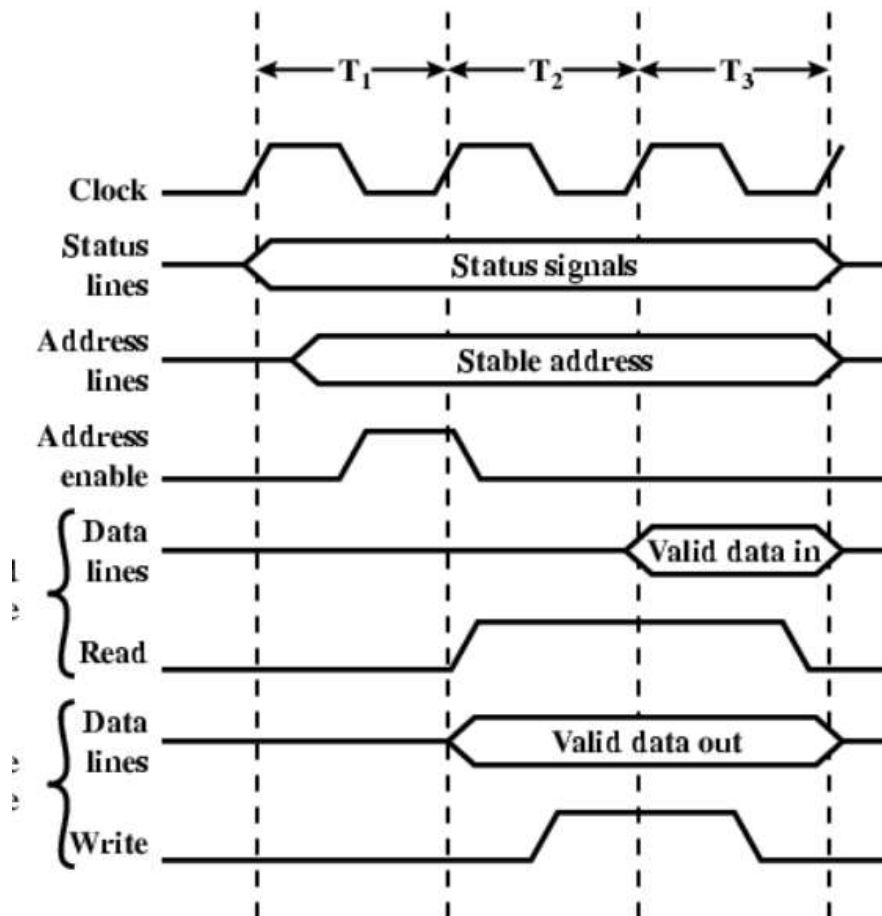
5.3.3. Temporización

Los buses están obligados a tener ciertos criterios de temporización. Esto se debe a que durante la comunicación entre componentes, cada vez que se levanta una señal que se envía a través del bus de interconexión, hay un tiempo que tiene que transcurrir hasta que la señal se estabilice y que la otra componente pueda garantizar que recibió la señal (este tiempo es especificado en la documentación de la componente en cuestión).

Existen dos tipos de sincronización de un bus, y cada bus puede optar por cualquiera de las dos: **temporización sincrónica** o **temporización asincrónica**.

Temporización sincrónica

En la temporización sincrónica, los eventos se coordina a través de un **reloj del bus**, que se asume universal a todas las componentes involucradas, a través de un protocolo cuidadoso que trabaja en relación a los ciclos de este reloj universal, y de esta manera se evita tener que revisar si los distintos eventos que forman parte de la comunicación efectivamente ocurrieron (no hay un “ponerse de acuerdo”, se mira únicamente el reloj). Además, todas los eventos ocurren en una cantidad fija y predeterminada de ciclos del reloj.



Aunque es fácil trabajar con buses síncronos debido a sus intervalos de tiempo discretos, también tienen algunos problemas. Por ejemplo, todo funciona en múltiplos del reloj del bus. Si una CPU y una memoria pueden completar una transferencia en 3,1 ciclos, tienen que estirla a 4,0 porque los ciclos fraccionarios están prohibidos.

Además, si un bus con temporización síncrona conecta dispositivos de distintas velocidades, algunos muy rápidos y otros más lentos, la velocidad del bus se tiene que ajustar al más lento, por lo tanto los dispositivos rápidos no pueden utilizar todo su potencial.

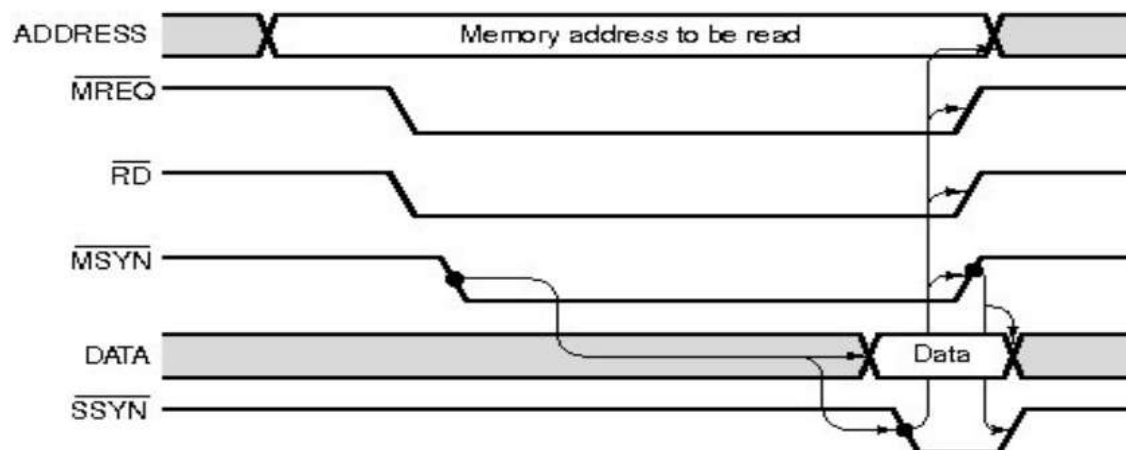
Temporización asincrónica

En la temporización asincrónica sigue habiendo un reloj involucrado (principalmente para coordinar los eventos internos de una componente en particular), pero la diferencia es que la coordinación de eventos entre distintas componentes no se encuentra regulada por el reloj del bus, sino que la ocurrencia de ciertos eventos dispara a otros eventos.

La idea es que se tiene un protocolo de transmisión de información, basado en que se comparten ciertas señales que permiten "ponerse de acuerdo" para identificar la etapa de la comunicación. Esto aumenta la complejidad en la administración del bus, pero permite una mayor flexibilidad en cuanto a las tareas que se pueden realizar con este tipo de buses.

Este tipo de temporización permite comunicaciones independientes a la cantidad de información transmitida, los eventos pueden tener cualquier longitud requerida, y no es necesario que los dispositivos tengan la misma velocidad. Además, resultan más eficientes en la transmisión de grandes cantidades de

información, por el hecho de que cada transferencia de palabra no tiene que ajustarse a una cantidad fija de ciclos de reloj.



5.3.4. Arbitraje

Durante la comunicación de dispositivos, algunos dispositivos que se conectan a un bus están activos y pueden iniciar transferencias de bus, mientras que otros son pasivos y esperan solicitudes. Los activos se llaman **maestros**, y los pasivos se llaman **esclavos**.

Cuando la CPU ordena a un controlador de disco que lea o escriba un bloque, la CPU actúa como maestro y el controlador de disco actúa como esclavo. Sin embargo, más adelante, el controlador de disco puede actuar como maestro cuando ordena a la memoria que acepte las palabras que está leyendo de la unidad de disco. En la figura 3-36 se enumeran varias combinaciones típicas de maestro y esclavo. Bajo ninguna circunstancia la memoria puede funcionar como maestro de la comunicación.

Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O device	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

Figure 3-36. Examples of bus masters and slaves.

Por lo tanto, es necesario realizar un **arbitraje** para poder decidir qué componente recibirá el acceso exclusivo (como maestro) al bus. Los mecanismos de arbitraje pueden ser **Centralizados** o **Distribuidos**.

Arbitraje Centralizado

En un arbitraje **Centralizado**, contamos con una componente electrónica específica que se encarga de determinar quién usa el bus. Muchas CPU tienen el árbitro integrado en el chip de la CPU, pero a veces se necesita un chip separado. En la figura 3-40 (a) se muestra un ejemplo sencillo de un bus de arbitraje centralizado.

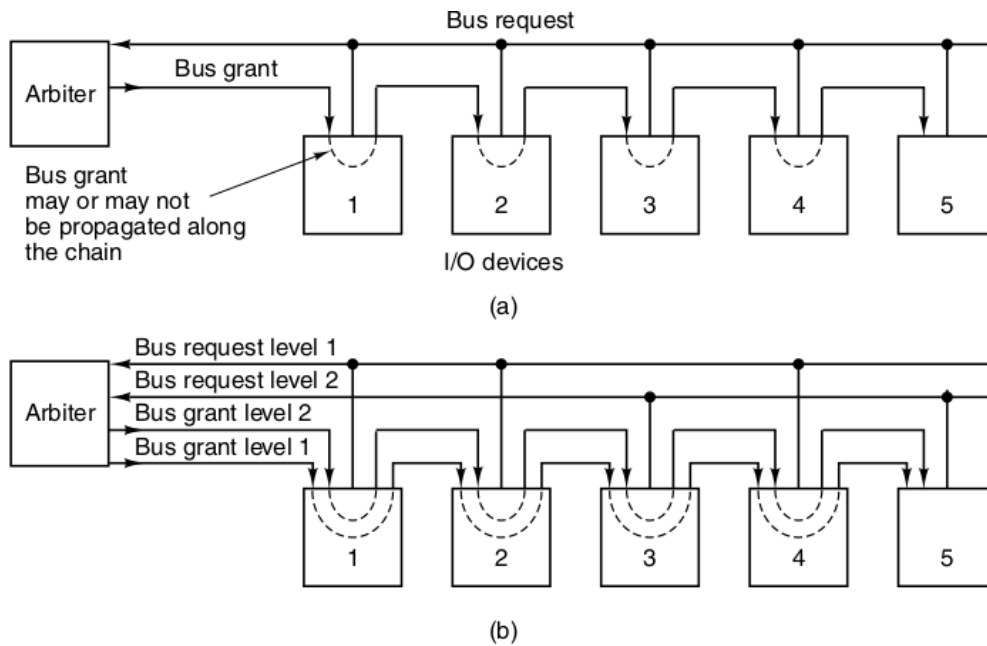


Figure 3-40. (a) A centralized one-level bus arbiter using daisy chaining. (b) The same arbiter, but with two levels.

El bus contiene una única línea de solicitud OR cableada que puede ser activada por uno o más dispositivos en cualquier momento. Cuando el árbitro ve una solicitud de bus, emite una señal de *grant*. Esta línea está conectada a través de todos los dispositivos de E/S en serie.

Este esquema se llama **daisy chaining**. Tiene la propiedad de que a los dispositivos se les asignan prioridades de manera efectiva según lo cerca que estén del árbitro. El dispositivo más cercano gana.

Para sortear las prioridades implícitas basadas en la distancia del árbitro, muchos buses tienen múltiples niveles de prioridad. Para cada nivel de prioridad hay una línea de solicitud de bus y una línea de *grant* de bus. El de la Fig. 3-40 (b) tiene dos niveles, 1 y 2. Cada dispositivo se conecta a uno de los niveles de solicitud de bus, y la idea es que los dispositivos de tiempo crítico se conecten a los de mayor prioridad.

En la figura 3-40 (b), los dispositivos 1, 2 y 4 usan la prioridad 1, mientras que los dispositivos 3 y 5 usan la prioridad 2. Si se solicitan varios niveles de prioridad al mismo tiempo, el árbitro emite una señal de *grant* solo en la que tenga la mayor prioridad. Entre los dispositivos de la misma prioridad, se utiliza el esquema de *daisy chaining*.

Arbitraje Distribuido

También es posible que el arbitraje de un bus sea distribuido. Por ejemplo, una computadora podría tener varias líneas de solicitud del bus (una por cada dispositivo), cada una con distinta prioridad (correspondiente al dispositivo). Cuando un dispositivo quiere usar el bus, envía una señal por su línea de solicitud. Todos los dispositivos monitorean todas las líneas de solicitud, por lo que al final de cada ciclo de bus cada dispositivo sabe si fue el solicitante de mayor prioridad, es decir que se le permite usar el bus durante el siguiente ciclo. En comparación con el arbitraje centralizado, este método de arbitraje requiere más líneas de bus pero evita el costo potencial de tener un circuito específico que actúe como árbitro.

Otro tipo de arbitraje distribuido, que se muestra en la figura 3-41, usa solo tres líneas, sin importar cuántos dispositivos estén presentes. La primera línea de bus es una línea OR cableada para solicitar el

bus. La segunda línea de bus se llama Busy, y la activa el actual maestro del bus. La tercera línea se utiliza para arbitrar el bus. Está conectado en cadena a través de todos los dispositivos. La cabeza de esta cadena se sostiene atándola a la fuente de alimentación.

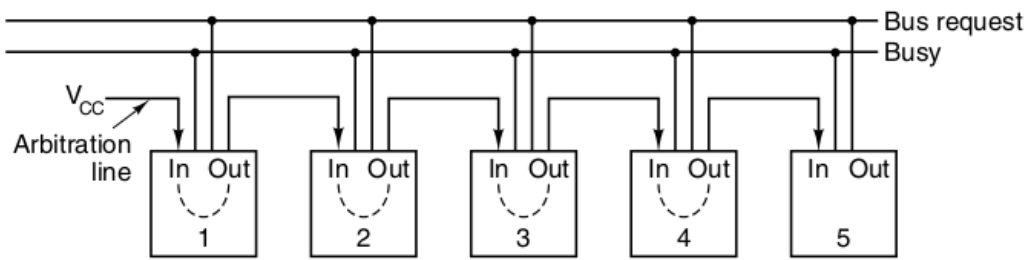


Figure 3-41. Decentralized bus arbitration.

Cuando ningún dispositivo desea el bus, la línea de arbitraje (activa) se propaga a todos los dispositivos. Para adquirir el bus, un dispositivo primero verifica si el bus está inactivo y si la señal de arbitraje IN que recibe está activa

Si IN está baja, significa que otro dispositivo de mayor prioridad está solicitando el bus, y por lo tanto no puede convertirse en maestro. Luego, replica una señal de OUT baja, para que el resto de los dispositivos de menor prioridad repliquen el mismo comportamiento.

Sin embargo, si IN está alta, y el dispositivo quiere el bus, el dispositivo baja la señal de OUT, lo que hace que su vecino de menor prioridad vea IN negado, y por lo tanto replique la señal negando su OUT.

Cuando las señales terminen de propagarse, solo un dispositivo tendrá su IN activo y su OUT negado. Este dispositivo se convierte en el maestro, activa la señal de BUSY y de OUT, y luego comienza su transferencia.

Notemos que el dispositivo más a la izquierda que quiere el bus lo obtiene. Por lo tanto, este esquema es similar al arbitraje en cadena original, con la diferencia de que no se cuenta con el árbitro, por lo que es más barato, más rápido y no está sujeto a fallas del árbitro.

Bibliografía

- [1] Herbert Bos Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, 2009.
- [2] Todd Austin Andrew S. Tanenbaum. *Structured Computer Organization*. Pearson, 2016.
- [3] John L. Hennessy David A. Patterson. *Computer Organization and Design: The Hardware/ Software Interface*. Morgan Kaufmann Publisher, 1993.
- [4] Linda Null y Julia Lobur. *The essentials of computer organization and architecture: Linda Null, Julia Lobur*. Jones y Bartlett, 2003.
- [5] William Stallings. *Computer Organization and Architecture: Designing for performance*. Pearson, 2006.